Top coder

Algorithm Tutorials

part #1

Step by step guide for getting professional
or
How to get closer to World Finals?

Perhaps you are already using C++ as your main programming language to solve TopCoder problems. This means that you have already used STL in a simple way, because arrays and strings are passed to your function as STL objects. You may have noticed, though, that many coders manage to write their code much more quickly and concisely than you.

Or perhaps you are not a C++ programmer, but want to become one because of the great functionality of this language and its libraries (and, maybe, because of the very short solutions you've read in TopCoder practice rooms and competitions).

Regardless of where you're coming from, this article can help. In it, we will review some of the powerful features of the Standard Template Library (STL) – a great tool that, sometimes, can save you a lot of time in an algorithm competition.

The simplest way to get familiar with STL is to begin from its containers.

Containers

Any time you need to operate with many elements you require some kind of container. In native C (not C++) there was only one type of container: the array.

The problem is not that arrays are limited (though, for example, it's impossible to determine the size of array at runtime). Instead, the main problem is that many problems require a container with greater functionality.

For example, we may need one or more of the following operations:

- Add some string to a container.
- Remove a string from a container.
- Determine whether a string is present in the container.
- Return a number of distinct elements in a container.
- Iterate through a container and get a list of added strings in some order.

Of course, one can implement this functionality in an ordinal array. But the trivial implementation would be very inefficient. You can create the tree- of hash- structure to solve it in a faster way, but think a bit: does the implementation of such a container depend on elements we are going to store? Do we have to re-implement the module to make it functional, for example, for points on a plane but not strings?

If not, we can develop the interface for such a container once, and then use everywhere for data of any type. That, in short, is the idea of STL containers.

Before we begin

When the program is using STL, it should #include the appropriate standard headers. For most containers the title of standard header matches the name of the container, and no extension is required. For example, if you are going to use stack, just add the following line at the beginning of your program:

#include <stack>

Container types (and algorithms, functors and all STL as well) are defined not in global namespace, but in special namespace called "std." Add the following line after your includes and before the code begin:

using namespace std;

Another important thing to remember is that the type of a container is the template parameter. Template parameters are specified with the '<'/'>' "brackets" in code. For example:

vector<int> N;

When making nested constructions, make sure that the "brackets" are not directly following one another – leave a blank between them.

vector< vector<int> > CorrectDefinition;

vector<vector<int>> WrongDefinition; // Wrong: compiler may be confused by 'operator >>'

Vector

The simplest STL container is vector. Vector is just an array with extended functionality. By the way, vector is the only container that is backward-compatible to native C code – this means that vector actually IS the array, but with some additional features.

```
vector<int> v(10);
for(int i = 0; i < 10; i++) {
     v[i] = (i+1)*(i+1);
}
for(int i = 9; i > 0; i--) {
     v[i] -= v[i-1];
}
```

Actually, when you type

```
vector<int> v;
```

the empty vector is created. Be careful with constructions like this:

```
vector<int> v[10];
```

Here we declare 'v' as an array of 10 vector<int>'s, which are initially empty. In most cases, this is not that we want. Use parentheses instead of brackets here. The most frequently used feature of vector is that it can report its size.

```
int elements_count = v.size();
```

Two remarks: first, size() is unsigned, which may sometimes cause problems. Accordingly, I usually define macros, something like sz(C) that returns size of C as ordinal signed int. Second, it's not a good practice to compare v.size() to zero if you want to know whether the container is empty. You're better off using empty() function:

```
bool is_nonempty_notgood = (v.size() >= 0); // Try to avoid this
bool is_nonempty_ok = !v.empty();
```

This is because not all the containers can report their size in O(1), and you definitely should not require counting all elements in a double-linked list just to ensure that it contains at least one.

Another very popular function to use in vector is push_back. Push_back adds an element to the end of vector, increasing its size by one. Consider the following example:

```
vector<int> v;
for(int i = 1; i < 1000000; i *= 2) {
     v.push_back(i);
}
int elements_count = v.size();
```

Don't worry about memory allocation -- vector will not allocate just one element each time. Instead, vector allocates more memory then it actually needs when adding new elements with push_back. The only thing you should worry about is memory usage, but at TopCoder this may not matter. (More on vector's memory policy later.)

When you need to resize vector, use the resize() function:

```
vector<int> v(20);
for(int i = 0; i < 20; i++) {
     v[i] = i+1;
}
v.resize(25);
for(int i = 20; i < 25; i++) {
     v[i] = i*2;
}
```

The resize() function makes vector contain the required number of elements. If you require less elements than vector already contain, the last ones will be deleted. If you ask vector to grow, it will enlarge its size and fill the newly created elements with zeroes.

Note that if you use push_back() after resize(), it will add elements AFTER the newly allocated size, but not INTO it. In the example above the size of the resulting vector is 25, while if we use push_back() in a second loop, it would be 30.

```
vector<int> v(20);
for(int i = 0; i < 20; i++) {
     v[i] = i+1;
}
v.resize(25);
for(int i = 20; i < 25; i++) {
     v.push_back(i*2); // Writes to elements with indices [25..30), not [20..25) ! <
}
```

To clear a vector use clear() member function. This function makes vector to contain 0 elements. It does not make elements zeroes -- watch out -- it completely erases the container.

There are many ways to initialize vector. You may create vector from another vector:

```
vector<int> v1;
// ...
vector<int> v2 = v1;
vector<int> v3(v1);
```

The initialization of v2 and v3 in the example above are exactly the same.

If you want to create a vector of specific size, use the following constructor:

```
vector<int> Data(1000);
```

In the example above, the data will contain 1,000 zeroes after creation. Remember to use parentheses, not brackets. If you want vector to be initialized with something else, write it in such manner:

```
vector<string> names(20, "Unknown");
```

Remember that you can create vectors of any type.

Multidimensional arrays are very important. The simplest way to create the two-dimensional array via vector is to create a vector of vectors.

```
vector< vector<int> > Matrix;
```

It should be clear to you now how to create the two-dimensional vector of given size:

```
int N, N;
// ...
```

```
vector< vector<int> > Matrix(N, vector<int>(M, -1));
```
Here we create a matrix of size N*M and fill it with -1.

The simplest way to add data to vector is to use push_back(). But what if we want to add data somewhere other than the end? There is the insert() member function for this purpose. And there is also the erase() member function to erase elements, as well. But first we need to say a few words about iterators.

You should remember one more very important thing: When vector is passed as a parameter to some function, a copy of vector is actually created. It may take a lot of time and memory to create new vectors when they are not really needed. Actually, it's hard to find a task where the copying of vector is REALLY needed when passing it as a parameter. So, you should never write:
```
void some_function(vector<int> v) { // Never do it unless you're sure what you do!
    // ...
}
```
Instead, use the following construction:
```
void some_function(const vector<int>& v) { // OK
    // ...
}
```
If you are going to change the contents of vector in the function, just omit the 'const' modifier.
```
int modify_vector(vector<int>& v) { // Correct
    V[0]++;
}
```
Pairs

Before we come to iterators, let me say a few words about pairs. Pairs are widely used in STL. Simple problems, like TopCoder SRM 250 and easy 500-point problems, usually require some simple data structure that fits well with pair. STL std::pair is just a pair of elements. The simplest form would be the following:
```
template<typename T1, typename T2> struct pair {
    T1 first;
    T2 second;
};
```
In general pair<int,int> is a pair of integer values. At a more complex level, pair<string, pair<int, int> > is a pair of string and two integers. In the second case, the usage may be like this:
```
pair<string, pair<int,int> > P;
string s = P.first; // extract string
int x = P.second.first; // extract first int
int y = P.second.second; // extract second int
```
The great advantage of pairs is that they have built-in operations to compare themselves. Pairs are compared first-to-second element. If the first elements are not equal, the result will be based on the comparison of the first elements only; the second elements will be compared only if the first ones are equal. The array (or vector) of pairs can easily be sorted by STL internal functions.

For example, if you want to sort the array of integer points so that they form a polygon, it's a good idea to put them to the vector< pair<double, pair<int,int> >, where each element of vector is { polar angle, { x, y } }. One call to the STL sorting function will give you the desired order of points.

Pairs are also widely used in associative containers, which we will speak about later in this article.

Iterators

What are iterators? In STL iterators are the most general way to access data in containers. Consider the simple problem: Reverse the array A of N int's. Let's begin from a C-like solution:
```
void reverse_array_simple(int *A, int N) {
    int first = 0, last = N-1; // First and last indices of elements to be swapped
    While(first < last) { // Loop while there is something to swap
        swap(A[first], A[last]); // swap(a,b) is the standard STL function
        first++; // Move first index forward
        last--; // Move last index back
    }
}
```
This code should be clear to you. It's pretty easy to rewrite it in terms of pointers:
```
void reverse_array(int *A, int N) {
    int *first = A, *last = A+N-1;
    while(first < last) {
        Swap(*first, *last);
        first++;
        last--;
    }
}
```
Look at this code, at its main loop. It uses only four distinct operations on pointers 'first' and 'last':

- compare pointers (first < last),
- get value by pointer (*first, *last),
- increment pointer, and
- decrement pointer

Now imagine that you are facing the second problem: Reverse the contents of a double-linked list, or a part of it. The first code, which uses indexing, will definitely not work. At least, it will not work in time, because it's impossible to get element by index in a double-linked list in O(1), only in O(N), so the whole algorithm will work in O(N^2). Errr...

But look: the second code can work for ANY pointer-like object. The only restriction is that that object can perform the operations described above: take value (unary *), comparison (<), and increment/decrement (++/--). Objects with these properties that are associated with containers are called iterators. Any STL container may be traversed by means of an iterator. Although not often needed for vector, it's very important for other container types.

So, what do we have? An object with syntax very much like a pointer. The following operations are defined for iterators:

- get value of an iterator, int x = *it;
- increment and decrement iterators it1++, it2--;
- compare iterators by '!=' and by '<'
- add an immediate to iterator it += 20; <=> shift 20 elements forward
- get the distance between iterators, int n = it2-it1;

But instead of pointers, iterators provide much greater functionality. Not only can they operate on any container, they may also perform, for example, range checking and profiling of container usage.

And the main advantage of iterators, of course, is that they greatly increase the reuse of code: your own algorithms, based on iterators, will work on a wide range of containers, and your own containers, which provide iterators, may be passed to a wide range of standard functions.

Not all types of iterators provide all the potential functionality. In fact, there are so-called "normal iterators" and "random access iterators". Simply put, normal iterators may be compared with '==' and '!=', and they may also be incremented and decremented. They may not be subtracted and we can not add a value to the normal iterator. Basically, it's impossible to implement the described operations in O(1) for all container types. In spite of this, the function that reverses array should look like this:

```
template<typename T> void reverse_array(T *first, T *last) {
     if(first != last) {
          while(true) {
               swap(*first, *last);
               first++;
               if(first == last) {
                    break;
               }
               last--;
               if(first == last) {
                    break;
               }
          }
     }
}
```

The main difference between this code and the previous one is that we don't use the "<" comparison on iterators, just the "==" one. Again, don't panic if you are surprised by the function prototype: template is just a way to declare a function, which works on any appropriate parameter types. This function should work perfectly on pointers to any object types and with all normal iterators.

Let's return to the STL. STL algorithms always use two iterators, called "begin" and "end." The end iterator is pointing not to the last object, however, but to the first invalid object, or the object directly following the last one. It's often very convenient.

Each STL container has member functions begin() and end() that return the begin and end iterators for that container.

Based on these principles, c.begin() == c.end() if and only if c is empty, and c.end() – c.begin() will always be equal to c.size(). (The last sentence is valid in cases when iterators can be subtracted, i.e. begin() and end() return random access iterators, which is not true for all kinds of containers. See the prior example of the double-linked list.)

The STL-compliant reverse function should be written as follows:

```
template<typename T> void reverse_array_stl_compliant(T *begin, T *end) {
     // We should at first decrement 'end'
```

```
        // But only for non-empty range
        if(begin != end)
        {
              end--;
              if(begin != end) {
                    while(true) {
                          swap(*begin, *end);
                          begin++;
                          If(begin == end) {
                                break;
                          }
                          end--;
                          if(begin == end) {
                                break;
                          }
                    }
              }
        }
  }
```

Note that this function does the same thing as the standard function std::reverse(T begin, T end) that can be found in algorithms module (#include <algorithm>).

In addition, any object with enough functionality can be passed as an iterator to STL algorithms and functions. That is where the power of templates comes in! See the following examples:

```
 vector<int> v;
 // ...
 vector<int> v2(v);
 vector<int> v3(v.begin(), v.end()); // v3 equals to v2

 int data[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 };
 vector<int> primes(data, data+(sizeof(data) / sizeof(data[0])));
```

The last line performs a construction of vector from an ordinal C array. The term 'data' without index is treated as a pointer to the beginning of the array. The term 'data + N' points to N-th element, so, when N if the size of array, 'data + N' points to first element not in array, so 'data + length of data' can be treated as end iterator for array 'data'. The expression 'sizeof(data)/sizeof(data[0])' returns the size of the array data, but only in a few cases, so don't use it anywhere except in such constructions. (C programmers will agree with me!)

Furthermore, we can even use the following constructions:

```
 vector<int> v;
 // ...
 vector<int> v2(v.begin(), v.begin() + (v.size()/2));
```

It creates the vector v2 that is equal to the first half of vector v.

Here is an example of reverse() function:

```
 int data[10] = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };
 reverse(data+2, data+6); // the range { 5, 7, 9, 11 } is now { 11, 9, 7, 5 };
```

Each container also has the rbegin()/rend() functions, which return reverse iterators. Reverse iterators are used to traverse the container in backward order. Thus:

```
 vector<int> v;
 vector<int> v2(v.rbegin()+(v.size()/2), v.rend());
```

will create v2 with first half of v, ordered back-to-front.

To create an iterator object, we must specify its type. The type of iterator can be constructed by a type of container by appending "::iterator", "::const_iterator", "::reverse_iterator" or "::const_reverse_iterator" to it. Thus, vector can be traversed in the following way:

```
vector<int> v;

 // ...

 // Traverse all container, from begin() to end()
 for(vector<int>::iterator it = v.begin(); it != v.end(); it++) {
      *it++; // Increment the value iterator is pointing to
 }
```

I recommend you use '!=' instead of '<', and 'empty()' instead of 'size() != 0' -- for some container types, it's just very inefficient to determine which of the iterators precedes another.

Now you know of STL algorithm reverse(). Many STL algorithms are declared in the same way: they get a pair of iterators – the beginning and end of a range – and return an iterator.

The find() algorithm looks for appropriate elements in an interval. If the element is found, the iterator pointing to the first occurrence of the element is returned. Otherwise, the return value equals the end of interval. See the code:

```
vector<int> v;
for(int i = 1; i < 100; i++) {
     v.push_back(i*i);
}

if(find(v.begin(), v.end(), 49) != v.end()) {
     // ...
}
```

To get the index of element found, one should subtract the beginning iterator from the result of find():

```
int i = (find(v.begin(), v.end(), 49) - v.begin();
if(i < v.size()) {
     // ...
}
```

Remember to #include <algorithm> in your source when using STL algorithms.

The min_element and max_element algorithms return an iterator to the respective element. To get the value of min/max element, like in find(), use *min_element(...) or *max_element(...), to get index in array subtract the begin iterator of a container or range:

```
int data[5] = { 1, 5, 2, 4, 3 };
vector<int> X(data, data+5);
int v1 = *max_element(X.begin(), X.end()); // Returns value of max element in vector
int i1 = min_element(X.begin(), X.end()) - X.begin; // Returns index of min element in vector

int v2 = *max_element(data, data+5); // Returns value of max element in array
int i3 = min_element(data, data+5) - data; // Returns index of min element in array
```

Now you may see that the useful macros would be:

```
#define all(c) c.begin(), c.end()
```

Don't put the whole right-hand side of these macros into parentheses -- that would be wrong!

Another good algorithm is sort(). It's very easy to use. Consider the following examples:

```
vector<int> X;

// ...

sort(X.begin(), X.end()); // Sort array in ascending order
sort(all(X)); // Sort array in ascending order, use our #define
sort(X.rbegin(), X.rend()); // Sort array in descending order using with reverse iterators
```

Compiling STL Programs

One thing worth pointing out here is STL error messages. As the STL is distributed in sources, and it becomes necessary for compilers to build efficient executables, one of STL's habits is unreadable error messages.

For example, if you pass a vector<int> as a const reference parameter (as you should do) to some function:

```
void f(const vector<int>& v) {
     for(
          vector<int>::iterator it = v.begin(); // hm... where's the error?..
          // ...
     // ...
}
```

The error here is that you are trying to create the non-const iterator from a const object with the begin() member function (though identifying that error can be harder than actually correcting it). The right code looks like this:

```
void f(const vector<int>& v) {
     int r = 0;
     // Traverse the vector using const_iterator
     for(vector<int>::const_iterator it = v.begin(); it != v.end(); it++) {
          r += (*it)*(*it);
     }
     return r;
}
```

In spite of this, let me tell about very important feature of GNU C++ called 'typeof'. This operator is replaced to the type of an expression during the compilation. Consider the following example:

```
typeof(a+b) x = (a+b);
```

This will create the variable x of type matching the type of (a+b) expression. Beware that typeof(v.size()) is unsigned for any STL container type. But the most important application of typeof for TopCoder is traversing a container. Consider the following macros:

```
#define tr(container, it) \
     for(typeof(container.begin()) it = container.begin(); it != container.end(); it++)
```

By using these macros we can traverse every kind of container, not only vector. This will produce const_iterator for const object and normal iterator for non-const object, and you will never get an error here.

```
void f(const vector<int>& v) {
      int r = 0;
      tr(v, it) {
            r += (*it)*(*it);
      }
      return r;
}
```

Note: I did not put additional parentheses on the #define line in order to improve its readability. See this article below for more correct #define statements that you can experiment with in practice rooms.

Traversing macros is not really necessary for vectors, but it's very convenient for more complex data types, where indexing is not supported and iterators are the only way to access data. We will speak about this later in this article.

Data manipulation in vector

One can insert an element to vector by using the insert() function:

```
vector<int> v;
// ...
v.insert(1, 42); // Insert value 42 after the first
```

All elements from second (index 1) to the last will be shifted right one element to leave a place for a new element. If you are planning to add many elements, it's not good to do many shifts – you're better off calling insert() one time. So, insert() has an interval form:

```
vector<int> v;
vector<int> v2;

// ..

// Shift all elements from second to last to the appropriate number of elements.
// Then copy the contents of v2 into v.
v.insert(1, all(v2));
```

Vector also has a member function erase, which has two forms. Guess what they are:

```
erase(iterator);
erase(begin iterator, end iterator);
```

At first case, single element of vector is deleted. At second case, the interval, specified by two iterators, is erased from vector.

The insert/erase technique is common, but not identical for all STL containers.

String

There is a special container to manipulate with strings. The string container has a few differences from vector<char>. Most of the differences come down to string manipulation functions and memory management policy.

String has a substring function without iterators, just indices:

```
string s = "hello";
string
      s1 = s.substr(0, 3), // "hel"
      s2 = s.substr(1, 3), // "ell"
      s3 = s.substr(0, s.length()-1), "hell"
      s4 = s.substr(1); // "ello"
```

Beware of (s.length()-1) on empty string because s.length() is unsigned and unsigned(0) – 1 is definitely not what you are expecting!

Set

It's always hard to decide which kind of container to describe first – set or map. My opinion is that, if the reader has a basic knowledge of algorithms, beginning from 'set' should be easier to understand.

Consider we need a container with the following features:

- add an element, but do not allow duples [duplicates?]
- remove elements
- get count of elements (distinct elements)
- check whether elements are present in set

This is quite a frequently used task. STL provides the special container for it – set. Set can add, remove and check the presence of particular element in O(log N), where N is the count of objects in the set. While adding elements to set, the duples [duplicates?] are discarded. A count of the elements in the set, N, is returned in O(1). We will speak of the algorithmic implementation of set and map later -- for now, let's investigate its interface:

```
set<int> s;

for(int i = 1; i <= 100; i++) {
     s.insert(i); // Insert 100 elements, [1..100]
}

s.insert(42); // does nothing, 42 already exists in set

for(int i = 2; i <= 100; i += 2) {
     s.erase(i); // Erase even values
}

int n = int(s.size()); // n will be 50
```

The push_back() member may not be used with set. It make sense: since the order of elements in set does not matter, push_back() is not applicable here.

Since set is not a linear container, it's impossible to take the element in set by index. Therefore, the only way to traverse the elements of set is to use iterators.

```
// Calculate the sum of elements in set
set<int> S;
// ...
int r = 0;
for(set<int>::const_iterator it = S.begin(); it != S.end(); it++) {
     r += *it;
}
```

It's more elegant to use traversing macros here. Why? Imagine you have a set< pair<string, pair< int, vector<int> > >. How to traverse it? Write down the iterator type name? Oh, no. Use our traverse macros instead.

```
set< pair<string, pair< int, vector<int> > > SS;
int total = 0;
tr(SS, it) {
     total += it->second.first;
}
```

Notice the 'it->second.first' syntax. Since 'it' is an iterator, we need to take an object from 'it' before operating. So, the correct syntax would be '(*it).second.first'. However, it's easier to write 'something->' than '(*something)'. The full explanation will be quite long –just remember that, for iterators, both syntaxes are allowed.

To determine whether some element is present in set use 'find()' member function. Don't be confused, though: there are several 'find()' 's in STL. There is a global algorithm 'find()', which takes two iterators, element, and works for O(N). It is possible to use it for searching for element in set, but why use an O(N) algorithm while there exists an O(log N) one? While searching in set and map (and also in multiset/multimap, hash_map/hash_set, etc.) do not use global find – instead, use member function 'set::find()'. As 'ordinal' find, set::find will return an iterator, either to the element found, or to 'end()'. So, the element presence check looks like this:

```
set<int> s;
// ...
if(s.find(42) != s.end()) {
     // 42 presents in set
}
else {
     // 42 not presents in set
}
```

Another algorithm that works for O(log N) while called as member function is count. Some people think that

```
if(s.count(42) != 0) {
     // …
}
```

or even

```
if(s.count(42)) {
     // …
}
```

is easier to write. Personally, I don't think so. Using count() in set/map is nonsense: the element either presents or not. As for me, I prefer to use the following two macros:

```
#define present(container, element) (container.find(element) != container.end())
#define cpresent(container, element) (find(all(container),element) != container.end())
```

(Remember that all(c) stands for "c.begin(), c.end()")

Here, 'present()' returns whether the element presents in the container with member function 'find()' (i.e. set/map, etc.) while 'cpresent' is for vector.

To erase an element from set use the erase() function.

```
 set<int> s;
 // …
 s.insert(54);
 s.erase(29);
The erase() function also has the interval form:
 set<int> s;
 // ..

 set<int>::iterator it1, it2;
 it1 = s.find(10);
 it2 = s.find(100);
 // Will work if it1 and it2 are valid iterators, i.e. values 10 and 100 present in set.
 s.erase(it1, it2); // Note that 10 will be deleted, but 100 will remain in the container
```

Set has an interval constructor:

```
int data[5] = { 5, 1, 4, 2, 3 };
 set<int> S(data, data+5);
```

It gives us a simple way to get rid of duplicates in vector, and sort it:

```
 vector<int> v;
 // …
 set<int> s(all(v));
 vector<int> v2(all(s));
```

Here 'v2' will contain the same elements as 'v' but sorted in ascending order and with duplicates removed.

Any comparable elements can be stored in set. This will be described later.

Map

There are two explanation of map. The simple explanation is the following:

```
 map<string, int> M;
 M["Top"] = 1;
 M["Coder"] = 2;
 M["SRM"] = 10;

 int x = M["Top"] + M["Coder"];

 if(M.find("SRM") != M.end()) {
      M.erase(M.find("SRM")); // or even M.erase("SRM")
 }
```

Very simple, isn't it?

Actually map is very much like set, except it contains not just values but pairs <key, value>. Map ensures that at most one pair with specific key exists. Another quite pleasant thing is that map has operator [] defined.

Traversing map is easy with our 'tr()' macros. Notice that iterator will be an std::pair of key and value. So, to get the value use it->second. The example follows:

```
map<string, int> M;
 // …
 int r = 0;
 tr(M, it) {
      r += it->second;
 }
```

Don't change the key of map element by iterator, because it may break the integrity of map internal data structure (see below).

There is one important difference between map::find() and map::operator []. While map::find() will never change the contents of map, operator [] will create an element if it does not exist. In some cases this could be very convenient, but it's definitly a bad idea to use operator [] many times in a loop, when you do not want to add new elements. That's why operator [] may not be used if map is passed as a const reference parameter to some function:

```
 void f(const map<string, int>& M) {
      if(M["the meaning"] == 42) { // Error! Cannot use [] on const map objects!
      }
      if(M.find("the meaning") != M.end() && M.find("the meaning")->second == 42) { // Correct
           cout << "Don't Panic!" << endl;
      }
 }
```

Notice on Map and Set

Internally map and set are almost always stored as red-black trees. We do not need to worry about the internal structure, the thing to remember is that the elements of map and set are always sorted in ascending order while traversing these containers. And that's why it's strongly not recommended to change the key value while traversing map or set: If you make the modification that breaks the order, it will lead

to improper functionality of container's algorithms, at least.

But the fact that the elements of map and set are always ordered can be practically used while solving TopCoder problems.

Another important thing is that operators ++ and -- are defined on iterators in map and set. Thus, if the value 42 presents in set, and it's not the first and the last one, than the following code will work:

```
set<int> S;
// ...
set<int>::iterator it = S.find(42);
set<int>::iterator it1 = it, it2 = it;
it1--;
it2++;
int a = *it1, b = *it2;
```

Here 'a' will contain the first neighbor of 42 to the left and 'b' the first one to the right.

More on algorithms

It's time to speak about algorithms a bit more deeply. Most algorithms are declared in the #include <algorithm> standard header. At first, STL provides three very simple algorithms: min(a,b), max(a,b), swap(a,b). Here min(a,b) and max(a,b) returns the minimum and maximum of two elements, while swap(a,b) swaps two elements.

Algorithm sort() is also widely used. The call to sort(begin, end) sorts an interval in ascending order. Notice that sort() requires random access iterators, so it will not work on all containers. However, you probably won't ever call sort() on set, which is already ordered.

You've already heard of algorithm find(). The call to find(begin, end, element) returns the iterator where 'element' first occurs, or end if the element is not found. Instead of find(...), count(begin, end, element) returns the number of occurrences of an element in a container or a part of a container. Remember that set and map have the member functions find() and count(), which works in O(log N), while std::find() and std::count() take O(N).

Other useful algorithms are next_permutation() and prev_permutation(). Let's speak about next_permutation. The call to next_permutation(begin, end) makes the interval [begin, end) hold the next permutation of the same elements, or returns false if the current permutation is the last one. Accordingly, next_permutation makes many tasks quite easy. If you want to check all permutations, just write:

```
vector<int> v;

for(int i = 0; i < 10; i++) {
     v.push_back(i);
}

do {
     Solve(..., v);
} while(next_permutation(all(v));
```

Don't forget to ensure that the elements in a container are sorted before your first call to next_permutation(...). Their initial state should form the very first permutation; otherwise, some permutations will not be checked.

String Streams

You often need to do some string processing/input/output. C++ provides two interesting objects for it: 'istringstream' and 'ostringstream'. They are both declared in #include <sstream>.

Object istringstream allows you to read from a string like you do from a standard input. It's better to view source:

```
void f(const string& s) {

     // Construct an object to parse strings
     istringstream is(s);

     // Vector to store data
     vector<int> v;

     // Read integer while possible and add it to the vector
     int tmp;
     while(is >> tmp) {
          v.push_back(tmp);
     }
}
```

The ostringstream object is used to do formatting output. Here is the code:

```
string f(const vector<int>& v) {

   // Constucvt an object to do formatted output
```

```
        ostringstream os;

        // Copy all elements from vector<int> to string stream as text
        tr(v, it) {
             os << ' ' << *it;
        }

        // Get string from string stream
        string s = os.str();

        // Remove first space character
        if(!s.empty()) { // Beware of empty string here
             s = s.substr(1);
        }

        return s;
 }
```

Summary

To go on with STL, I would like to summarize the list of templates to be used. This will simplify the reading of code samples and, I hope, improve your TopCoder skills. The short list of templates and macros follows:

```
 typedef vector<int> vi;
 typedef vector<vi> vvi;
 typedef pair<int,int> ii;
 #define sz(a) int((a).size())
 #define pb push_back
 #defile all(c) (c).begin(),(c).end()
 #define tr(c,i) for(typeof((c).begin() i = (c).begin(); i != (c).end(); i++)
 #define present(c,x) ((c).find(x) != (c).end())
 #define cpresent(c,x) (find(all(c),x) != (c).end())
```

The container vector<int> is here because it's really very popular. Actually, I found it convenient to have short aliases to many containers (especially for vector<string>, vector<ii>, vector< pair<double, ii> >). But this list only includes the macros that are required to understand the following text.

Another note to keep in mind: When a token from the left-hand side of #define appears in the right-hand side, it should be placed in braces to avoid many nontrivial problems.

Creating Vector from Map

As you already know, map actually contains pairs of element. So you can write it in like this:

```
map<string, int> M;
 // ...
 vector< pair<string, int> > V(all(M)); // remember all(c) stands for
 (c).begin(),(c).end()
```

Now vector will contain the same elements as map. Of course, vector will be sorted, as is map. This feature may be useful if you are not planning to change elements in map any more but want to use indices of elements in a way that is impossible in map.

Copying data between containers

Let's take a look at the copy(...) algorithm. The prototype is the following:

```
copy(from_begin, from_end, to_begin);
```

This algorithm copies elements from the first interval to the second one. The second interval should have enough space available. See the following code:

```
 vector<int> v1;
 vector<int> v2;

 // ...

 // Now copy v2 to the end of v1
 v1.resize(v1.size() + v2.size());
 // Ensure v1 have enough space
 copy(all(v2), v1.end() - v2.size());
 // Copy v2 elements right after v1 ones
```

Another good feature to use in conjunction with copy is inserters. I will not describe it here due to limited space but look at the code:

```
 vector<int> v;
 // ...
 set<int> s;
 // add some elements to set
```

```
 copy(all(v), inserter(s));
The last line means:
 tr(v, it) {
// remember traversing macros from Part I
      s.insert(*it);
 }
```

But why use our own macros (which work only in gcc) when there is a standard function? It's a good STL practice to use standard algorithms like copy, because it will be easy to others to understand your code.

To insert elemements to vector with push_back use back_inserter, or front_inserter is available for deque container. And in some cases it is useful to remember that the first two arguments for 'copy' may be not only begin/end, but also rbegin/rend, which copy data in reverse order.

Merging lists
Another common task is to operate with sorted lists of elements. Imagine you have two lists of elements -- A and B, both ordered. You want to get a new list from these two. There are four common operations here:

- 'union' the lists, R = A+B
- intersect the lists, R = A*B
- set difference, R = A*(~B) or R = A-B
- set symmetric difference, R = A XOR B

STL provides four algorithms for these tasks: set_union(...), set_intersection(...), set_difference(...) and set_symmetric_difference(...). They all have the same calling conventions, so let's look at set_intersection. A free-styled prototype would look like this:
end_result = set_intersection(begin1, end1, begin2, end2, begin_result);
Here [begin1,end1) and [begin2,end2) are the input lists. The 'begin_result' is the iterator from where the result will be written. But the size of the result is unknown, so this function returns the end iterator of output (which determines how many elements are in the result). See the example for usage details:

```
int data1[] = { 1, 2, 5, 6, 8, 9, 10 };
int data2[] = { 0, 2, 3, 4, 7, 8, 10 };

vector<int> v1(data1, data1+sizeof(data1)/sizeof(data1[0]));
vector<int> v2(data2, data2+sizeof(data2)/sizeof(data2[0]));

vector<int> tmp(max(v1.size(), v2.size());

vector<int> res = vector<int> (tmp.begin(), set_intersection(all(v1), all(v2), tmp.begin());
```

Look at the last line. We construct a new vector named 'res'. It is constructed via interval constructor, and the beginning of the interval will be the beginning of tmp. The end of the interval is the result of the set_intersection algorithm. This algorithm will intersect v1 and v2 and write the result to the output iterator, starting from 'tmp.begin()'. Its return value will actually be the end of the interval that forms the resulting dataset.

One comment that might help you understand it better: If you would like to just get the number of elements in set intersection, use int cnt = set_intersection(all(v1), all(v2), tmp.begin()) – tmp.begin();

Actually, I would never use a construction like ' vector<int> tmp'. I don't think it's a good idea to allocate memory for each set_*** algorithm invoking. Instead, I define the global or static variable of appropriate type and enough size. See below:

```
set<int> s1, s2;
for(int i = 0; i < 500; i++) {
        s1.insert(i*(i+1) % 1000);
        s2.insert(i*i*i % 1000);
}

static int temp[5000]; // greater than we need

vector<int> res = vi(temp, set_symmetric_difference(all(s1), all(s2), temp));
int cnt = set_symmetric_difference(all(s1), all(s2), temp) - temp;
```

Here 'res' will contain the symmetric difference of the input datasets.

Remember, input datasets need to be sorted to use these algorithms. So, another important thing to remember is that, because sets are always ordered, we can use set-s (and even map-s, if you are not scared by pairs) as parameters for these algorithms.

These algorithms work in single pass, in O(N1+N2), when N1 and N2 are sizes of input datasets.

Calculating Algorithms

Yet another interesting algorithm is accumulate(...). If called for a vector of int-s and third parameter zero, accumulate(...) will return the sum of elements in vector:

```
vector<int> v;
// ...
int sum = accumulate(all(v), 0);
The result of accumulate() call always has the type of its third argument. So, if you are not
sure that the sum fits in integer, specify the third parameter's type directly:
vector<int> v;
// ...
long long sum = accumulate(all(v), (long long)0);
```

Accumulate can even calculate the product of values. The fourth parameter holds the predicate to use in calculations. So, if you want the product:

```
vector<int> v;
// ...
double product = accumulate(all(v), double(1), multiplies<double>());
// don't forget to start with 1 !
Another interesting algorithm is inner_product(...). It calculates the scalar product of two
intervals. For example:
vector<int> v1;
vector<int> v2;
for(int i = 0; i < 3; i++) {
      v1.push_back(10-i);
      v2.push_back(i+1);
}
int r = inner_product(all(v1), v2.begin(), 0);
```

'r' will hold (v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2]), or (10*1+9*2+8*3), which is 52.

As for 'accumulate' the type of return value for inner_product is defined by the last parameter. The last parameter is the initial value for the result. So, you may use inner_product for the hyperplane object in multidimensional space: just write inner_product(all(normal), point.begin(), -shift).

It should be clear to you now that inner_product requires only increment operation from iterators, so queues and sets can also be used as parameters. Convolution filter, for calculating the nontrivial median value, could look like this:

```
set<int> values_ordered_data(all(data));
int n = sz(data); // int n = int(data.size());
vector<int> convolution_kernel(n);
for(int i = 0; i < n; i++) {
     convolution_kernel[i] = (i+1)*(n-i);
}
double result = double(inner_product(all(ordered_data), convolution_kernel.begin(), 0)) /
accumulate(all(convolution_kernel), 0);
```

Of course, this code is just an example -- practically speaking, it would be faster to copy values to another vector and sort it.

It's also possible to write a construction like this:

```
vector<int> v;
// ...
int r = inner_product(all(v), v.rbegin(), 0);
```

This will evaluate V[0]*V[N-1] + V[1]+V[N-2] + ... + V[N-1]*V[0] where N is the number of elements in 'v'.

Nontrivial Sorting

Actually, sort(...) uses the same technique as all STL:

- all comparison is based on 'operator <'

This means that you only need to override 'operator <'. Sample code follows:

```
struct fraction {
      int n, d; // (n/d)
      // ...
      bool operator < (const fraction& f) const {
           if(false) {
                return (double(n)/d) < (double(f.n)/f.d);
                // Try to avoid this, you're the TopCoder!
           }
           else {
```

```
                return n*f.d < f.n*d;
            }
        }
};

// ...

vector<fraction> v;

// ...

sort(all(v));
```

In cases of nontrivial fields, your object should have default and copy constructor (and, maybe, assignment operator -- but this comment is not for TopCoders).

Remember the prototype of 'operator <' : return type bool, const modifier, parameter const reference.

Another possibility is to create the comparison functor. Special comparison predicate may be passed to the sort(...) algorithm as a third parameter. Example: sort points (that are pair<double,double>) by polar angle.

```
typedef pair<double, double> dd;

const double epsilon = 1e-6;

struct sort_by_polar_angle {
       dd center;
       // Constuctor of any type
       // Just find and store the center
       template<typename T> sort_by_polar_angle(T b, T e) {
            int count = 0;
            center = dd(0,0);
            while(b != e) {
                        center.first += b->first;
                        center.second += b->second;
                  b++;
                count++;
            }
                  double k = count ? (1.0/count) : 0;
            center.first *= k;
                  center.second *= k;
       }
// Compare two points, return true if the first one is earlier
// than the second one looking by polar angle
// Remember, that when writing comparator, you should
// override not 'operator <' but 'operator ()'
       bool operator () (const dd& a, const dd& b) const {
            double p1 = atan2(a.second-center.second, a.first-center.first);
            double p2 = atan2(b.second-center.second, b.first-center.first);
            return p1 + epsilon < p2;
       }
};

// ...

vector<dd> points;

// ...

       sort(all(points), sort_by_polar_angle(all(points)));
```

This code example is complex enough, but it does demonstrate the abilities of STL. I should point out that, in this sample, all code will be inlined during compilation, so it's actually really fast.

Also remember that 'operator <' should always return false for equal objects. It's very important – for the reason why, see the next section.

Using your own objects in Maps and Sets

Elements in set and map are ordered. It's the general rule. So, if you want to enable using of your objects in set or map you should make them comparable. You already know the rule of comparisons in STL:

| * all comparison is based on 'operator <'

Again, you should understand it in this way: "I only need to implement operator < for objects to be stored in set/map."

Imagine you are going to make the 'struct point' (or 'class point'). We want to intersect some line segments and make a set of intersection points (sound familiar?). Due to finite computer precision, some points will be the same while their coordinates differ a bit. That's what you should write:

```
const double epsilon = 1e-7;


struct point {
       double x, y;

       // ...


       // Declare operator < taking precision into account
       bool operator < (const point& p) const {
            if(x < p.x - epsilon) return true;
            if(x > p.x + epsilon) return false;
            if(y < p.y - epsilon) return true;
            if(y > p.y + epsilon) return false;
            return false;
       }
 };
```

Now you can use set<point> or map<point, string>, for example, to look up whether some point is already present in the list of intersections. An even more advanced approach: use map<point, vector<int> > and list the list of indices of segments that intersect at this point.

It's an interesting concept that for STL 'equal' does not mean 'the same', but we will not delve into it here.

Memory management in Vectors

As has been said, vector does not reallocate memory on each push_back(). Indeed, when push_back() is invoked, vector really allocates more memory than is needed for one additional element. Most STL implementations of vector double in size when push_back() is invoked and memory is not allocated. This may not be good in practical purposes, because your program may eat up twice as much memory as you need. There are two easy ways to deal with it, and one complex way to solve it.

The first approach is to use the reserve() member function of vector. This function orders vector to allocate additional memory. Vector will not enlarge on push_back() operations until the size specified by reserve() will be reached.

Consider the following example. You have a vector of 1,000 elements and its allocated size is 1024. You are going to add 50 elements to it. If you call push_back() 50 times, the allocated size of vector will be 2048 after this operation. But if you write

```
v.reserve(1050);
```

before the series of push_back(), vector will have an allocated size of exactly 1050 elements.

If you are a rapid user of push_back(), then reserve() is your friend.

By the way, it's a good pattern to use v.reserve() followed by copy(…, back_inserter(v)) for vectors.

Another situation: after some manipulations with vector you have decided that no more adding will occur to it. How do you get rid of the potential allocation of additional memory? The solution follows:

```
vector<int> v;
// ...
vector<int>(all(v)).swap(v);
```

This construction means the following: create a temporary vector with the same content as v, and then swap this temporary vector with 'v'. After the swap the original oversized v will be disposed. But, most likely, you won't need this during SRMs.

The proper and complex solution is to develop your own allocator for the vector, but that's definitely not a topic for a TopCoder STL tutorial.

Implementing real algorithms with STL

Armed with STL, let's go on to the most interesting part of this tutorial: how to implement real algorithms efficiently.

Depth-first search (DFS)

I will not explain the theory of DFS here – instead, read this section of gladius's Introduction to Graphs and Data Structures tutorial – but I will show you how STL can help.

At first, imagine we have an undirected graph. The simplest way to store a graph in STL is to use the lists of vertices adjacent to each vertex. This leads to the vector< vector<int> > W structure, where W[i] is a list of vertices adjacent to i. Let's verify our graph is connected via DFS:

```
/*
Reminder from Part 1:
typedef vector<int> vi;
typedef vector<vi> vvi;
*/

 int N; // number of vertices
 vvi W; // graph
 vi V; // V is a visited flag

 void dfs(int i) {
       if(!V[i]) {
            V[i] = true;
            for_each(all(W[i]), dfs);
       }
 }

 bool check_graph_connected_dfs() {
       int start_vertex = 0;
       V = vi(N, false);
       dfs(start_vertex);
       return (find(all(V), 0) == V.end());
 }
```

That's all. STL algorithm 'for_each' calls the specified function, 'dfs', for each element in range. In check_graph_connected() function we first make the Visited array (of correct size and filled with zeroes). After DFS we have either visited all vertices, or not – this is easy to determine by searching for at least one zero in V, by means of a single call to find().

Notice on for_each: the last argument of this algorithm can be almost anything that "can be called like a function". It may be not only global function, but also adapters, standard algorithms, and even member functions. In the last case, you will need mem_fun or mem_fun_ref adapters, but we will not touch on those now.

One note on this code: I don't recommend the use of vector<bool>. Although in this particular case it's quite safe, you're better off not to use it. Use the predefined 'vi' (vector<int>). It's quite OK to assign true and false to int's in vi. Of course, it requires 8*sizeof(int)=8*4=32 times more memory, but it works well in most cases and is quite fast on TopCoder.

A word on other container types and their usage

Vector is so popular because it's the simplest array container. In most cases you only require the functionality of an array from vector – but, sometimes, you may need a more advanced container.

It is not good practice to begin investigating the full functionality of some STL container during the heat of a Single Round Match. If you are not familiar with the container you are about to use, you'd be better off using vector or map/set. For example, stack can always be implemented via vector, and it's much faster to act this way if you don't remember the syntax of stack container.

STL provides the following containers: list, stack, queue, deque, priority_queue. I've found list and deque quite useless in SRMs (except, probably, for very special tasks based on these containers). But queue and priority_queue are worth saying a few words about.

Queue

Queue is a data type that has three operations, all in O(1) amortized: add an element to front (to "head") remove an element from back (from "tail") get the first unfetched element ("tail") In other words, queue is the FIFO buffer.

Breadth-first search (BFS)

Again, if you are not familiar with the BFS algorithm, please refer back to this TopCoder tutorial first. Queue is very convenient to use in BFS, as shown below:

```
/*
Graph is considered to be stored as adjacent vertices list.
Also we considered graph undirected.

vvi is vector< vector<int> >
W[v] is the list of vertices adjacent to v
*/

 int N; // number of vertices
 vvi W; // lists of adjacent vertices
```

```
bool check_graph_connected_bfs() {
      int start_vertex = 0;
      vi V(N, false);
      queue<int> Q;
      Q.push(start_vertex);
      V[start_vertex] = true;
      while(!Q.empty()) {
            int i = Q.front();
            // get the tail element from queue
            Q.pop();
            tr(W[i], it) {
                  if(!V[*it]) {
                        V[*it] = true;
                        Q.push(*it);
                  }
            }
      }
      return (find(all(V), 0) == V.end());
}
```

More precisely, queue supports front(), back(), push() (== push_back()), pop (== pop_front()). If you also need push_front() and pop_back(), use deque. Deque provides the listed operations in O(1) amortized.

There is an interesting application of queue and map when implementing a shortest path search via BFS in a complex graph. Imagine that we have the graph, vertices of which are referenced by some complex object, like:
pair< pair<int,int>, pair< string, vector< pair<int, int> > > >

(this case is quite usual: complex data structure may define the position in some game, Rubik's cube situation, etc…)

Consider we know that the path we are looking for is quite short, and the total number of positions is also small. If all edges of this graph have the same length of 1, we could use BFS to find a way in this graph. A section of pseudo-code follows:

```
// Some very hard data structure

typedef pair< pair<int,int>, pair< string, vector< pair<int, int> > > > POS;

// ...

int find_shortest_path_length(POS start, POS finish) {

     map<POS, int> D;
     // shortest path length to this position
     queue<POS> Q;

     D[start] = 0; // start from here
     Q.push(start);

     while(!Q.empty()) {
          POS current = Q.front();
          // Peek the front element
          Q.pop(); // remove it from queue

          int current_length = D[current];

          if(current == finish) {
               return D[current];
               // shortest path is found, return its length
          }

          tr(all possible paths from 'current', it) {
               if(!D.count(*it)) {
               // same as if(D.find(*it) == D.end), see Part I
                    // This location was not visited yet
                    D[*it] = current_length + 1;
               }
          }
     }
```

```
    // Path was not found
    return -1;
}

// ...
```

If the edges have different lengths, however, BFS will not work. We should use Dijkstra instead. It's possible to implement such a Dijkstra via priority_queue -- see below.

Priority_Queue
Priority queue is the binary heap. It's the data structure, that can perform three operations:

- push any element (push)
- view top element (top)
- pop top element (pop)

For the application of STL's priority_queue see the TrainRobber problem from SRM 307.

Dijkstra
In the last part of this tutorial I'll describe how to efficiently implement Dijktra's algorithm in sparse graph using STL containers. Please look through this tutorial for information on Dijkstra's algoritm.

Consider we have a weighted directed graph that is stored as vector< vector< pair<int,int> > > G, where

- G.size() is the number of vertices in our graph
- G[i].size() is the number of vertices directly reachable from vertex with index i
- G[i][j].first is the index of j-th vertex reachable from vertex i
- G[i][j].second is the length of the edge heading from vertex i to vertex G[i][j].first

We assume this, as defined in the following two code snippets:

```
typedef pair<int,int> ii;
typedef vector<ii> vii;
typedef vector<vii> vvii;
```

Dijkstra via priority_queue
Many thanks to misof for spending the time to explain to me why the complexity of this algorithm is good despite not removing deprecated entries from the queue.

```
    vi D(N, 987654321);
    // distance from start vertex to each vertex

    priority_queue<ii,vector<ii>, greater<ii> > Q;
    // priority_queue with reverse comparison operator,
    // so top() will return the least distance
    // initialize the start vertex, suppose it's zero
    D[0] = 0;
    Q.push(ii(0,0));

    // iterate while queue is not empty
    while(!Q.empty()) {

        // fetch the nearest element
        ii top = Q.top();
        Q.pop();

        // v is vertex index, d is the distance
        int v = top.second, d = top.first;

        // this check is very important
        // we analyze each vertex only once
        // the other occurrences of it on queue (added earlier)
        // will have greater distance
        if(d <= D[v]) {
            // iterate through all outcoming edges from v
            tr(G[v], it) {
                int v2 = it->first, cost = it->second;
```

```
                        if(D[v2] > D[v] + cost) {
                              // update distance if possible
                              D[v2] = D[v] + cost;
                              // add the vertex to queue
                              Q.push(ii(D[v2], v2));

                        }
                  }
            }
      }
```

I will not comment on the algorithm itself in this tutorial, but you should notice the priority_queue object definition. Normally, priority_queue<ii> will work, but the top() member function will return the largest element, not the smallest. Yes, one of the easy solutions I often use is just to store not distance but (-distance) in the first element of a pair. But if you want to implement it in the "proper" way, you need to reverse the comparison operation of priority_queue to reverse one. Comparison function is the third template parameter of priority_queue while the second paramerer is the storage type for container. So, you should write priority_queue<ii, vector<ii>, greater<ii> >.

Dijkstra via set

Petr gave me this idea when I asked him about efficient Dijkstra implementation in C#. While implementing Dijkstra we use the priority_queue to add elements to the "vertices being analyzed" queue in O(logN) and fetch in O(log N). But there is a container besides priority_queue that can provide us with this functionality -- it's 'set'! I've experimented a lot and found that the performance of Dijkstra based on priority_queue and set is the same.

So, here's the code:

```
      vi D(N, 987654321);

      // start vertex
      set<ii> Q;
      D[0] = 0;
      Q.insert(ii(0,0));

      while(!Q.empty()) {

            // again, fetch the closest to start element
            // from "queue" organized via set
            ii top = *Q.begin();
            Q.erase(Q.begin());
            int v = top.second, d = top.first;

            // here we do not need to check whether the distance
            // is perfect, because new vertices will always
            // add up in proper way in this implementation

            tr(G[v], it) {
                  int v2 = it->first, cost = it->second;
                  if(D[v2] > D[v] + cost) {
                        // this operation can not be done with priority_queue,
                        // because it does not support DECREASE_KEY
                        if(D[v2] != 987654321) {
                              Q.erase(Q.find(ii(D[v2],v2)));
                        }
                        D[v2] = D[v] + cost;
                        Q.insert(ii(D[v2], v2));
                  }
            }
      }
```

One more important thing: STL's priority_queue does not support the DECREASE_KEY operation. If you will need this operation, 'set' may be your best bet.

I've spent a lot of time to understand why the code that removes elements from queue (with set) works as fast as the first one.

These two implementations have the same complexity and work in the same time. Also, I've set up practical experiments and the performance is exactly the same (the difference is about ~%0.1 of time).

As for me, I prefer to implement Dijkstra via 'set' because with 'set' the logic is simpler to understand, and we don't need to remember about 'greater<int>' predicate overriding.

What is not included in STL

If you have made it this far in the tutorial, I hope you have seen that STL is a very powerful tool, especially for TopCoder SRMs. But before you embrace STL wholeheartedly, keep in mind what is NOT included in it.

First, STL does not have BigInteger-s. If a task in an SRM calls for huge calculations, especially multiplication and division, you have three options:

- use a pre-written template
- use Java, if you know it well
- say "Well, it was definitely not my SRM!"

I would recommend option number one.

Nearly the same issue arises with the geometry library. STL does not have geometry support, so you have those same three options again.

The last thing – and sometimes a very annoying thing – is that STL does not have a built-in string splitting function. This is especially annoying, given that this function is included in the default template for C++ in the ExampleBuilder plugin! But actually I've found that the use of istringstream(s) in trivial cases and sscanf(s.c_str(), …) in complex cases is sufficient.

Those caveats aside, though, I hope you have found this tutorial useful, and I hope you find the STL a useful addition to your use of C++. Best of luck to you in the Arena!

Note from the author: In both parts of this tutorial I recommend the use of some templates to minimize the time required to implement something. I must say that this suggestion should always be up to the coder. Aside from whether templates are a good or bad tactic for SRMs, in everyday life they can become annoying for other people who are trying to understand your code. While I did rely on them for some time, ultimately I reached the decision to stop. I encourage you to weigh the pros and cons of templates and to consider this decision for yourself.

Prime numbers and their properties were extensively studied by the ancient Greek mathematicians. Thousands of years later, we commonly use the different properties of integers that they discovered to solve problems. In this article we'll review some definitions, well-known theorems, and number properties, and look at some problems associated with them.

A prime number is a positive integer, which is divisible on 1 and itself. The other integers, greater than 1, are **composite**. **Coprime** integers are a set of integers that have no common divisor other than 1 or -1.

The fundamental theorem of arithmetic:
Any positive integer can be divided in primes in essentially only one way. The phrase 'essentially one way' means that we do not consider the order of the factors important.

One is neither a prime nor composite number. One is not composite because it doesn't have two distinct divisors. If one is prime, then number 6, for example, has two different representations as a product of prime numbers: 6 = 2 * 3 and 6 = 1 * 2 * 3. This would contradict the fundamental theorem of arithmetic.

Euclid's theorem:
There is no largest prime number.

To prove this, let's consider only n prime numbers: $p_1, p_2, \ldots, p_n$. But no prime p_i divides the number

$N = p_1 * p_2 * \ldots * p_n + 1$,

so N cannot be composite. This contradicts the fact that the set of primes is finite.

Exercise 1. Sequence a_n is defined recursively:

$$a_1 = 2,\ a_{n+1} = a_n^2 - a_n + 1$$

Prove that a_i and a_j, $i \neq j$ are relatively prime.

Hint: Prove that $a_{n+1} = a_1 a_2 \ldots a_n + 1$ and use Euclid's theorem.

Exercise 2. Ferma numbers F_n ($n \geq 0$) are positive integers of the form

$$F_n = 2^{2^n} + 1$$

Prove that F_i and F_j, $i \neq j$ are relatively prime.

Hint: Prove that $F_{n+1} = F_0 F_1 F_2 \ldots F_n + 2$ and use Euclid's theorem.

Dirichlet's theorem about arithmetic progressions:
For any two positive coprime integers a and b there are infinitely many primes of the form $a + n*b$, where $n > 0$.

Trial division:
Trial division is the simplest of all factorization techniques. It represents a brute-force method, in which we are trying to divide n by every number i not greater than the square root of n. (Why don't we need to test values larger than the square root of n?) The procedure *factor* prints the factorization of number n. The factors will be printed in a line, separated with one space. The number n can contain no more than one factor, greater than n.

```
void factor(int n)
{
  int i;
  for(i=2;i<=(int)sqrt(n);i++)
  {
    while(n % i == 0)
    {
```

```
        printf("%d ",i);
        n /= i;
      }
    }
    if (n > 1) printf("%d",n);
    printf("\n");
  }
```

Consider a problem that asks you to find the factorization of integer $g(-2^{31} < g < 2^{31})$ in the form

$g = f_1 \times f_2 \times \ldots \times f_n$ or $g = -1 \times f_1 \times f_2 \times \ldots \times f_n$

where f_i is a prime greater than 1 and $f_i \le f_j$ for $i < j$.

For example, for g = -192 the answer is -192 = -1 x 2 x 2 x 2 x 2 x 2 x 2 x 3.

To solve the problem, it is enough to use trial division as shown in function *factor*.

Sieve of Eratosthenes:
The most efficient way to find all small primes was proposed by the Greek mathematician Eratosthenes. His idea was to make a list of positive integers not greater than *n* and sequentially strike out the multiples of primes less than or equal to the square root of *n*. After this procedure only primes are left in the list.

The procedure of finding prime numbers *gen_primes* will use an array primes[MAX] as a list of integers. The elements of this array will be filled so that

$$\text{primes}[i] = \begin{cases} 1, \text{if } i \text{ is prime} \\ 0, \text{if } i \text{ is composite} \end{cases}$$

At the beginning we mark all numbers as prime. Then for each prime number i ($i \ge 2$), not greater than √MAX, we mark all numbers $i*i$, $i*(i+1)$, … as composite.

```
void gen_primes()
{
  int i,j;
  for(i=0;i<MAX;i++) primes[i] = 1;
  for(i=2;i<=(int)sqrt(MAX);i++)
    if (primes[i])
      for(j=i;j*i<MAX;j++) primes[i*j] = 0;
}
```

For example, if MAX = 16, then after calling *gen_primes*, the array 'primes' will contain next values:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
primes[*i*]	1	1	1	1	0	1	0	1	0	0	0	1	0	1	0	0

Goldbach's Conjecture:
For any integer n ($n \ge 4$) there exist two prime numbers p_1 and p_2 such that $p_1 + p_2 = n$. In a problem we might need to find the number of essentially different pairs (p_1, p_2), satisfying the condition in the conjecture for a given even number n ($4 \le n \le 2\ 15$). (The word 'essentially' means that for each pair (p_1, p_2) we have $p_1 \le p_2$.)

For example, for $n = 10$ we have two such pairs: 10 = 5 + 5 and 10 = 3 + 7.

To solve this,as $n \le 2^{15} = 32768$, we'll fill an array primes[32768] using function *gen_primes*. We are interested in primes, not greater than 32768.

The function *FindSol*(n) finds the number of different pairs (p_1, p_2), for which $n = p_1 + p_2$. As $p_1 \le p_2$, we have $p_1 \le n/2$. So to solve the problem we need to find the number of pairs $(i, n - i)$, such that i and $n - i$ are prime numbers and $2 \le i \le n/2$.

```
int FindSol(int n)
{
  int i,res=0;
  for(i=2;i<=n/2;i++)
    if (primes[i] && primes[n-i]) res++;
  return res;
}
```

Euler's totient function

The number of positive integers, not greater than *n*, and relatively prime with *n*, equals to Euler's totient function ϕ (*n*). In symbols we can state that

ϕ (*n*) ={*a* Î N: 1 ≤ *a* ≤ *n*, gcd(*a*, *n*) = 1}

This function has the following properties:

1. If *p* is prime, then ϕ (*p*) = *p* – 1 and ϕ (p^a) = *p a* * (1 – 1/*p*) for any *a*.
2. If *m* and *n* are coprime, then ϕ (*m* * *n*) = ϕ (*m*) * ϕ (*n*).
3. If *n* = $p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$, then Euler function can be found using formula:

ϕ (*n*) = *n* * (1 – 1/*p* 1) * (1 – 1/*p* 2) * ... * (1 – 1/*p k*)

The function fi(*n*) finds the value of ϕ(*n*):

```
int fi(int n)
{
  int result = n;
  for(int i=2;i*i <= n;i++)
  {
    if (n % i == 0) result -= result / i;
    while (n % i == 0) n /= i;
  }
  if (n > 1) result -= result / n;
  return result;
}
```

For example, to find ϕ(616) we need to factorize the argument: 616 = 2^3 * 7 * 11. Then, using the formula, we'll get:

ϕ(616) = 616 * (1 – 1/2) * (1 – 1/7) * (1 – 1/11) = 616 * 1/2 * 6/7 * 10/11 = 240.

Say you've got a problem that, for a given integer *n* (0 < *n* ≤ 10^9), asks you to find the number of positive integers less than *n* and relatively prime to *n*. For example, for *n* = 12 we have 4 such numbers: 1, 5, 7 and 11.

The solution: The number of positive integers less than *n* and relatively prime to *n* equals to ϕ(*n*). In this problem, then, we need do nothing more than to evaluate Euler's totient function.

Or consider a scenario where you are asked to calculate a function Answer(*x*, *y*), with *x* and *y* both integers in the range [1, *n*], 1 ≤ *n* ≤ 50000. If you know Answer(*x*, *y*), then you can easily derive Answer(*k**x*, *k**y*) for any integer *k*. In this situation you want to know how many values of Answer(*x*, *y*) you need to precalculate. The function Answer is not symmetric.

For example, if *n* = 4, you need to precalculate 11 values: Answer(1, 1), Answer(1, 2), Answer(2, 1), Answer(1, 3), Answer(2, 3), Answer(3, 2), Answer(3, 1), Answer(1, 4), Answer(3, 4), Answer(4, 3) and Answer(4, 1).

The solution here is to let res(*i*) be the minimum number of Answer(*x*, *y*) to precalculate, where *x*, *y* Î{1, …, *i*}. It is obvious that res(1) = 1, because if *n* = 1, it is enough to know Answer(1, 1). Let we know res(*i*). So for *n* = *i* + 1 we need to find Answer(1, *i* + 1), Answer(2, *i* + 1), … , Answer(*i* + 1, *i* + 1), Answer(*i* + 1, 1), Answer(*i* + 1, 2), … , Answer(*i* + 1, *i*).

The values Answer(*j*, *i* + 1) and Answer(*i* + 1, *j*), *j* Î{1, …, *i* + 1}, can be found from known values if GCD(*j*, *i* + 1) > 1, i.e. if the numbers *j* and *i* + 1 are not common primes. So we must know all the values Answer(*j*, *i* + 1) and Answer(*i* + 1, *j*) for which *j* and *i* + 1 are coprime. The number of such values equals to 2 * ϕ (*i* + 1), where ϕ is an Euler's totient function. So we have a recursion to solve a problem:

```
res(1) = 1,
res(i + 1) = res(i) + 2 * j (i + 1), i > 1
```

Euler's totient theorem:
If n is a positive integer and a is coprime to n, then $a^{\phi(n)} \equiv 1 \pmod{n}$.

Fermat's little theorem:
If p is a prime number, then for any integer a that is coprime to n, we have

$a^{p} \equiv a \pmod{p}$

This theorem can also be stated as: If p is a prime number and a is coprime to p, then

$a^{p-1} \equiv 1 \pmod{p}$

Fermat's little theorem is a special case of Euler's totient theorem when n is prime.

The number of divisors:
If $n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$, then the number of its positive divisors equals to

$(a_1 + 1) * (a_2 + 1) * \ldots * (a_k + 1)$

For a proof, let A i be the set of divisors $\{1, p_i, p_i^2, \ldots, p_i^{a_i}\}, 1 \le i \le k$. Any divisor of number n can be represented as a product $x_1 * x_2 * \ldots * x_k$, where $x_i \in A_i$. As $|A_i| = a_i + 1$, we have

$(a_1 + 1) * (a_2 + 1) * \ldots * (a_k + 1)$

possibilities to get different products $x_1 * x_2 * \ldots * x_k$.

For example, to find the number of divisors for 36, we need to factorize it first: $36 = 2^2 * 3^2$. Using the formula above, we'll get the divisors amount for 36. It equals to $(2 + 1) * (2 + 1) = 3 * 3 = 9$. There are 9 divisors for 36: 1, 2, 3, 4, 6, 9, 12, 18 and 36.

Here's another problem to think about: For a given positive integer n ($0 < n < 2^{31}$) we need to find the number of such m that $1 \le m \le n$, GCD(m, n) $\ne 1$ and GCD(m, n) $\ne m$. For example, for $n = 6$ we have only one such number $m = 4$.

The solution is to subtract from n the amount of numbers, coprime with it (its amount equals to $\phi(n)$) and the amount of its divisors. But the number 1 simultaneously is coprime with n and is a divisor of n. So to obtain the difference we must add 1. If $n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$ is a factorization of n, the number n has $(a_1 + 1) * (a_2 + 1) * \ldots * (a_k + 1)$ divisors. So the answer to the problem for a given n equals to

$n - \phi(n) - (a_1 + 1) * (a_2 + 1) * \ldots * (a_k + 1) + 1$

Practice Room:
Want to put some of these theories into practice? Try out these problems, from the TopCoder Archive:

- Refactoring (SRM 216)
- PrimeAnagrams (SRM 223)
- DivisibilityCriteria (SRM 239)
- PrimePolynom (SRM 259)
- DivisorInc (SRM 302)
- PrimePalindromic (SRM 303)
- RugSizes (SRM 304)
- PowerCollector (SRM 305)

Recursion is a wonderful programming tool. It provides a simple, powerful way of approaching a variety of problems. It is often hard, however, to see how a problem can be approached recursively; it can be hard to "think" recursively. It is also easy to write a recursive program that either takes too long to run or doesn't properly terminate at all. In this article we'll go over the basics of recursion and hopefully help you develop, or refine, a very important programming skill.

What is Recursion?
In order to say exactly what recursion is, we first have to answer "What is recursion?" Basically, a function is said to be recursive if it calls itself. Below is pseudocode for a recursive function that prints the phrase "Hello World" a total of *count* times:

```
function HelloWorld(count)
{
    if(count<1)return
    print("Hello World!")
    HelloWorld(count - 1)
}
```

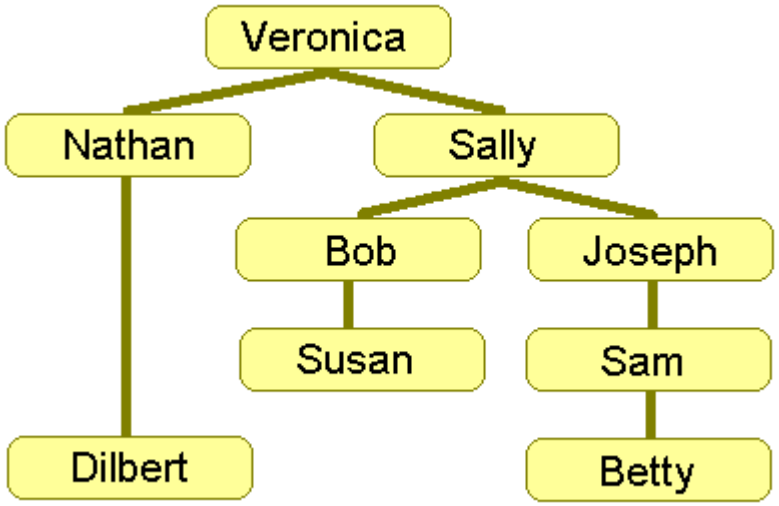
It might not be immediately clear what we're doing here - so let's follow through what happens if we call our function with *count* set to 10. Since *count* is not less than 1, we do nothing on the first line. On the next, we print "Hello World!" once. At this point we need to print our phrase 9 more times. Since we now have a HelloWorld function that can do just that, we simply call HelloWorld (this time with *count* set to 9) to print the remaining copies. That copy of HelloWorld will print the phrase once, and then call another copy of HelloWorld to print the remaining 8. This will continue until finally we call HelloWorld with *count* set to zero. HelloWorld(0) does nothing; it just returns. Once HelloWorld(0) has finished, HelloWorld(1) is done too, and it returns. This continues all the way back to our original call of HelloWorld(10), which finishes executing having printed out a total of 10 "Hello World!"s.

You may be thinking this is not terribly exciting, but this function demonstrates some key considerations in designing a recursive algorithm:

1. **It handles a simple "base case" without using recursion.**
   In this example, the base case is "HelloWorld(0)"; if the function is asked to print zero times then it returns without spawning any more "HelloWorld"s.
2. **It avoids cycles.**
   Imagine if "HelloWorld(10)" called "HelloWorld(10)" which called "HelloWorld(10)." You'd end up with an infinite cycle of calls, and this usually would result in a "stack overflow" error while running. In many recursive programs, you can avoid cycles by having each function call be for a problem that is somehow smaller or simpler than the original problem. In this case, for example, *count* will be smaller and smaller with each call. As the problem gets simpler and simpler (in this case, we'll consider it "simpler" to print something zero times rather than printing it 5 times) eventually it will arrive at the "base case" and stop recursing. There are many ways to avoid infinite cycles, but making sure that we're dealing with progressively smaller or simpler problems is a good rule of thumb.
3. **Each call of the function represents a complete handling of the given task.**
   Sometimes recursion can seem kind of magical in the way it breaks down big problems. However, there is no such thing as a free lunch. When our function is given an argument of 10, we print "Hello World!" once and then we print it 9 more times. We can pass a part of the job along to a recursive call, but the original function still has to account for all 10 copies somehow.

Why use Recursion?
The problem we illustrated above is simple, and the solution we wrote works, but we probably would have been better off just using a loop instead of bothering with recursion. Where recursion tends to shine is in situations where the problem is a little more complex. Recursion can be applied to pretty much any problem, but there are certain scenarios for which you'll find it's particularly helpful. In the remainder of this article we'll discuss a few of these scenarios and, along the way, we'll discuss a few more core ideas to keep in mind when using recursion.

Scenario #1: Hierarchies, Networks, or Graphs
In algorithm discussion, when we talk about a graph we're generally not talking about a chart showing the relationship between variables (like your TopCoder ratings graph, which shows the relationship between time and your rating). Rather, we're usually talking about a network of things, people, or concepts that are connected to each other in various ways. For example, a road map could be thought of as a graph that shows cities and how they're connected by roads. Graphs can be large, complex, and awkward to deal with programatically. They're also very common in algorithm theory and algorithm competitions. Luckily, working with graphs can be made much simpler using recursion. One common type of a graph is a hierarchy, an example of which is a business's organization chart:



Name	Manager
Betty	Sam
Bob	Sally
Dilbert	Nathan
Joseph	Sally
Nathan	Veronica
Sally	Veronica
Sam	Joseph
Susan	Bob
Veronica	

In this graph, the objects are people, and the connections in the graph show who reports to whom in the company. An upward line on our graph says that the person lower on the graph reports to the person above them. To the right we see how this structure could be represented in a database. For each employee we record their name and the name of their manager (and from this information we could rebuild the whole hierarchy if required - do you see how?).

Now suppose we are given the task of writing a function that looks like "countEmployeesUnder(employeeName)". This function is intended to tell us how many employees report (directly or indirectly) to the person named by *employeeName*. For example, suppose we're calling "countEmployeesUnder('Sally')" to find out how many employees report to Sally.

To start off, it's simple enough to count how many people work directly under her. To do this, we loop through each database record, and for each employee whose manager is Sally we increment a counter variable. Implementing this approach, our function would return a count of 2: Bob and Joseph. This is a start, but we also want to count people like Susan or Betty who are lower in the hierarchy but report to Sally indirectly. This is awkward because when looking at the individual record for Susan, for example, it's not immediately clear how Sally is involved.

A good solution, as you might have guessed, is to use recursion. For example, when we encounter Bob's record in the database we don't just increment the counter by one. Instead, we increment by one (to count Bob) and then increment it by the number of people who report to Bob. How do we find out how many people report to Bob? We use a recursive call to the function we're writing: "countEmployeesUnder('Bob')". Here's pseudocode for this approach:

```
function countEmployeesUnder(employeeName)
{
    declare variable counter
    counter = 0
    for each person in employeeDatabase
    {
        if(person.manager == employeeName)
        {
            counter = counter + 1
            counter = counter + countEmployeesUnder(person.name)
        }
    }
    return counter
}
```

If that's not terribly clear, your best bet is to try following it through line-by-line a few times mentally. Remember that each time you make a recursive call, you get a new copy of all your local variables. This means that there will be a separate copy of *counter* for each call. If that wasn't the case, we'd really mess things up when we set counter to zero at the beginning of the function. As an exercise, consider how we could change the function to increment a global variable instead. Hint: if we were incrementing a global variable, our function wouldn't need to return a value.

Mission Statements

A very important thing to consider when writing a recursive algorithm is to have a clear idea of our function's "mission statement." For example, in this case I've assumed that a person shouldn't be counted as reporting to him or herself. This means "countEmployeesUnder('Betty')" will return zero. Our function's mission statment might thus be "Return the count of people who report, directly or indirectly, to the person named in *employeeName* - not including the person named *employeeName*."

Let's think through what would have to change in order to make it so a person did count as reporting to him or herself. First off, we'd need to make it so that if there are no people who report to someone we return one instead of zero. This is simple -- we just change the line "counter = 0" to "counter = 1" at the beginning of the function. This makes sense, as our function has to return a value 1 higher than it did before. A call to "countEmployeesUnder('Betty')" will now return 1.

However, we have to be very careful here. We've changed our function's mission statement, and when working with recursion that means taking a close look at how we're using the call recursively. For example, "countEmployeesUnder('Sam')" would now give an incorrect answer of 3. To see why, follow through the code: First, we'll count Sam as 1 by setting counter to 1. Then when we encounter Betty we'll count her as 1. Then we'll count the employees who report to Betty -- and that will return 1 now as well.

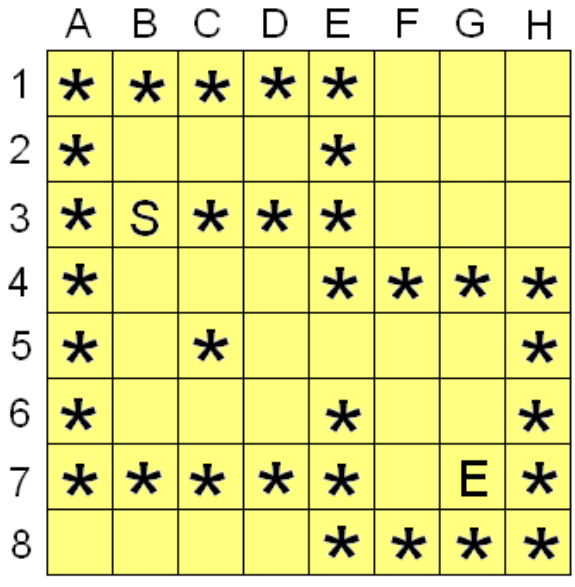
It's clear we're double counting Betty; our function's "mission statement" no longer matches how we're using it. We need to get rid of the line "counter = counter + 1", recognizing that the recursive call will now count Betty as "someone who reports to Betty" (and thus we don't need to count her before the recursive call).

As our functions get more and more complex, problems with ambiguous "mission statements" become more and more apparent. In order to make recursion work, we must have a very clear specification of what each function call is doing or else we can end up with some very difficult to debug errors. Even if time is tight it's often worth starting out by writing a comment detailing exactly what the function is supposed to do. Having a clear "mission statement" means that we can be confident our recursive calls will behave as we expect and the whole picture will come together correctly.

Scenario #2: Multiple Related Decisions

When our program only has to make one decision, our approach can be fairly simple. We loop through each of the options for our decision, evaluate each one, and pick the best. If we have two decisions, we can have nest one loop inside the other so that we try each possible combination of decisions. However, if we have a lot of decisions to make (possibly we don't even know how many decisions we'll need to make), this approach doesn't hold up.

For example, one very common use of recursion is to solve mazes. In a good maze we have multiple options for which way to go. Each of those options may lead to new choices, which in turn may lead to new choices as the path continues to branch. In the process of getting from start to finish, we may have to make a number of related decisions on which way to turn. Instead of making all of these decisions at once, we can instead make just one decision. For each option we try for the first decision, we then make a recursive call to try each possibility for all of the remaining decisions. Suppose we have a maze like this:



For this maze, we want to determine the following: is it possible to get from the 'S' to the 'E' without passing through any '*' characters. The function call we'll be handling is something like this: "isMazeSolveable(maze[ ][ ])". Our maze is represented as a 2 dimensional array of characters, looking something like the grid above. Now naturally we're looking for a recursive solution, and indeed we see our basic "multiple related decision" pattern here. To solve our maze we'll try each possible initial decision (in this case we start at B3, and can go to B2 or B4), and then use recursion to continue exploring each of those initial paths. As we keep recursing we'll explore further and further from the start. If the maze is solveable, at some point we'll reach the 'E' at G7. That's one of our base cases: if we are asked "can we get from G7 to the end", we'll see that we're already at the end and return true without further recursion. Alternatively, if we can't get to the end from either B2 or B4, we'll know that we can't get to the end from B3 (our initial starting point) and thus we'll return false.

Our first challenge here is the nature of the input we're dealing with. When we make our recursive call, we're going to want an easy way to specify where to start exploring from - but the only parameter we've been passed is the maze itself. We could try moving the 'S' character around in the maze in order to tell each recursive call where to start. That would work, but would be very slow because in each call we'd have to first look through the entire maze to find where the 'S' is. A better idea would be to find the 'S' once, and then pass around our starting point in separate variables. This happens fairly often when using recursion: we have to use a "starter" function that will initialize any data and get the parameters in a form that will be easy to work with. Once things are ready, the "starter" function calls the recursive function that will do the rest of the work. Our starter function here might look something like this:

```
function isMazeSolveable(maze[][])
{
    declare variables x,y,startX,startY
    startX=-1
    startY=-1

    // Look through grid to find our starting point
    for each x from A to H
    {
        for each y from 1 to 8
        {
            if maze[x][y]=='S' then
            {
                startX=x
                startY=y
            }
        }
    }

    // If we didn't find starting point, maze isn't solveable
    if startX==-1 then return false

    // If we did find starting point, start exploring from that point
    return exploreMaze(maze[][],startX,startY)
}
```

We're now free to write our recursive function exploreMaze. Our mission statement for the function will be "Starting at the position (X,Y), is it possible to reach the 'E' character in the given maze. If the position (x,y) is not traversable, then return false." Here's a first stab at the code:

```
function exploreMaze(maze[][],x,y)
{
    // If the current position is off the grid, then
    // we can't keep going on this path
    if y>8 or y<1 or x<'A' or x>'H' then return false

    // If the current position is a '*', then we
    // can't continue down this path
    if maze[x][y]=='*' then return false

    // If the current position is an 'E', then
    // we're at the end, so the maze is solveable.
    if maze[x][y]=='E' then return true

    // Otherwise, keep exploring by trying each possible
    // next decision from this point.  If any of the options
    // allow us to solve the maze, then return true.  We don't
    // have to worry about going off the grid or through a wall -
    // we can trust our recursive call to handle those possibilities
    // correctly.
```

```
    if exploreMaze(maze,x,y-1) then return true  // search up
    if exploreMaze(maze,x,y+1) then return true  // search down
    if exploreMaze(maze,x-1,y) then return true  // search left
    if exploreMaze(maze,x+1,y) then return true  // search right

    // None of the options worked, so we can't solve the maze
    // using this path.
    return false
}
```

Avoiding Cycles

If you're keen eyed, you likely noticed a flaw in our code above. Consider what happens when we're exploring from our initial position of B3. From B3, we'll try going up first, leading us to explore B2. From there, we'll try up again and go to B1. B1 won't work (there's a '*' there), so that will return false and we'll be back considering B2. Since up didn't work, we'll try down, and thus we'll consider B3. And from B3, we'll consider B2 again. This will continue on until we error out: there's an infinite cycle.

We've forgotten one of our rules of thumb: we need to make sure the problem we're considering is somehow getting smaller or simpler with each recursive call. In this case, testing whether we can reach the end from B2 is no simpler than considering whether we can reach the end from B3. Here we can get a clue from real-life mazes: if you feel like you've seen this place before, then you may be going in circles. We need to revise our mission statement to include "avoid exploring from any position we've already considered". As the number of places we've considered grows, the problem gets simpler and simpler because each decision will have less valid options.

The remaining problem is, then, "how do we keep track of places we've already considered?". A good solution would be to pass around another 2 dimensional array of true/false values that would contain a "true" for each grid cell we've already been to. A quicker-and-dirtier way would be to change *maze* itself, replacing the current position with a '*' just before we make any recursive calls. This way, when any future path comes back to the point we're considering, it'll know that it went in a circle and doesn't need to continue exploring. Either way, we need to make sure we mark the current point as visited before we make the recursive calls, as otherwise we won't avoid the infinite cycle.

Scenario #3: Explicit Recursive Relationships

You may have heard of the Fibonacci number sequence. This sequence looks like this: 0, 1, 1, 2, 3, 5, 8, 13... After the first two values, each successive number is the sum of the previous two numbers. We can define the Fibonacci sequence like this:

```
    Fibonacci[0] = 0
    Fibonacci[1] = 1
    Fibonacci[n] = Fibonacci[n-2] + Fibonacci[n-1]
```

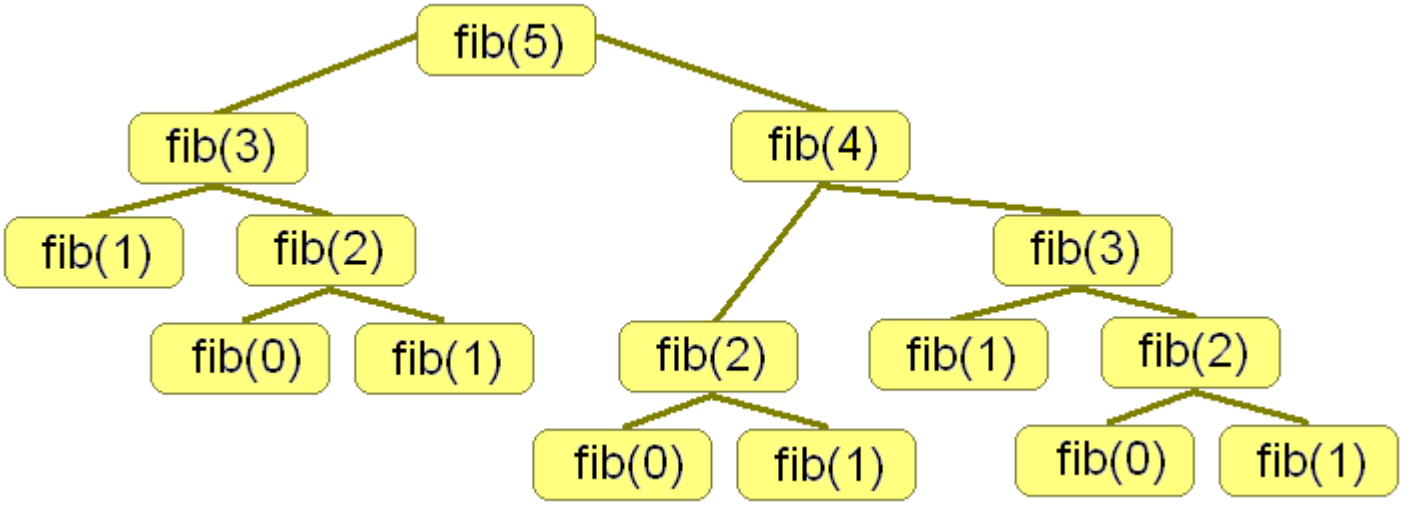
This definition already looks a lot like a recursive function. 0 and 1 are clearly the base cases, and the other possible values can be handled with recursion. Our function might look like this:

```
function fib(n)
{
    if(n<1)return 0
    if(n==1)return 1
    return fib(n-2) + fib(n-1)
}
```

This kind of relationship is very common in mathematics and computer science - and using recursion in your software is a very natural way to model this kind of relationship or sequence. Looking at the above function, our base cases (0 and 1) are clear, and it's also clear that *n* gets smaller with each call (and thus we shouldn't have problems with infinite cycles this time).

Using a Memo to Avoid Repetitious Calculation

The above function returns correct answers, but in practice it is extremely slow. To see why, look at what happens if we called "fib(5)". To calculate "fib(5)", we'll need to calculate "fib(4)" and "fib(3)". Each of these two calls will make two recursive calls each - and they in turn will spawn more calls. The whole execution tree will look like this:



The above tree grows exponentially for higher values of *n* because of the way calls tend to split - and because of the tendency we have to keep re-calculating the same values. In calculating "fib(5)", we ended up calculating "fib(2)" 3 times. Naturally, it would be better to only calculate that value once - and then remember that value so that it doesn't need to be calculated again next time it is asked for. This is the basic idea of memoization. When we calculate an answer, we'll store it in an array (named *memo* for this example) so we can reuse that answer later. When the function is called, we'll first check to see if we've already got the answer stored in *memo*, and if we do we'll return that value immediately instead of recalculating it.

To start off, we'll initialize all the values in *memo* to -1 to mark that they have not been calculated yet. It's convenient to do this by making a "starter" function and a recursive function like we did before:

```
function fib(n)
{
    declare variable i,memo[n]

    for each i from 0 to n
    {
        memo[i]=-1
    }
    memo[0]=0
    memo[1]=1

    return calcFibonacci(n,memo)
}

function calcFibonacci(n,memo)
{
    // If we've got the answer in our memo, no need to recalculate
    if memo[n]!=-1 then return memo[n]

    // Otherwise, calculate the answer and store it in memo
    memo[n] = calcFibonacci(n-2,memo) + calcFibonacci(n-1,memo)

    // We still need to return the answer we calculated
    return memo[n]
}
```

The execution tree is now much smaller because values that have been calculated already no longer spawn more recursive calls. The result is that our program will run much faster for larger values of *n*. If our program is going to calculate a lot of Fibonacci numbers, it might be best to keep *memo* somewhere more persistent; that would save us even more calculations on future calls. Also, you might have noticed another small trick in the above code. Instead of worrying about the base cases inside *calcFibonacci*, we pre-loaded values for those cases into the memo. Pre-loading base values - especially if there's a lot of them - can make our recursive functions faster by allowing us to check base cases and the memo at the same time. The difference is especially noticeable in situations where the base cases are more numerous or hard to distinguish.

This basic memoization pattern can be one of our best friends in solving TopCoder algorithm problems. Often, using a memo is as simple as looking at the input parameters, creating a *memo* array that corresponds to those input parameters, storing calculated values at the end of the function, and checking the memo as the function starts. Sometimes the input parameters won't be simple integers that map easily to a *memo* array - but by using other objects (like a hash table) for the *memo* we can continue with the same general pattern. In general, if you find a recursive solution for a problem, but find that the solution runs too slowly, then the solution is often memoization.

Conclusion

Recursion is a fundamental programming tool that can serve you well both in TopCoder competitions and "real world" programming. It's a subject that many experienced programmers still find threatening, but practice using recursion in TopCoder situations will give you a great start in thinking recursively, and using recursion to solve complicated programming problems.

As a programmer, you'll frequently come across tasks that deal with a number of objects -- numbers, strings, people, and so forth -- and that require you to store and process those objects. If you need to maintain a list of objects that are sorted and unique, and if you need to be able to quickly insert and retrieve objects to and from this list, the ideal data structure will be a tree set (or a tree map, if you consider each object a key and associate another object called a value to it).

Many programming languages provide built-in support for tree-based sets and maps -- for instance, Java's TreeSet/TreeMap classes and the C++ Standard Template Library's set and map classes -- but because of their common use, it's easy to misunderstand what actually happens behind the scenes, and how the underlying data structures actually work. That's what this article is all about.

We'll start off by looking at some of the general terms and concepts used in dealing with trees We'll then focus on binary search trees (BST), a special type of tree that keeps elements sorted (but doesn't guarantee efficient insertion and retrieval). Finally we'll look at red-black trees, a variation of binary search trees that overcome BST's limitations through a logarithmic bound on insertion and retrieval.

Trees terminology

A tree is a data structure that represents data in a hierarchical manner. It associates every object to a node in the tree and maintains the parent/child relationships between those nodes. Each tree must have exactly one node, called the **root**, from which all nodes of the tree extend (and which has no parent of its own). The other end of the tree – the last level down -- contains the leaf nodes of the tree.

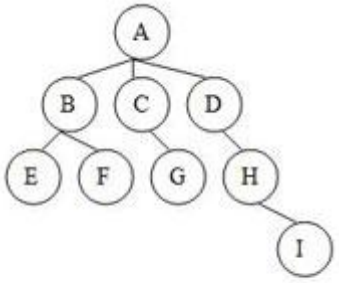


Figure 1. A tree

The number of lines you pass through when you travel from the root until you reach a particular node is the **depth** of that node in the tree (node G in the figure above has a depth of 2). The **height** of the tree is the maximum depth of any node in the tree (the tree in Figure 1 has a height of 3). The number of children emanating from a given node is referred to as its **degree** -- for example, node A above has a degree of 3 and node H has a degree of 1.

Binary Search Tree (BST)

A binary search tree is a tree with one additional constraint -- it keeps the elements in the tree in a particular order. Formally each node in the BST has two children (if any are missing we consider it a **nil node**), a left child and a right child. Nodes are rooted in place based on their values, with the smallest on the left and largest on the right.

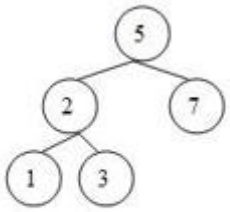


Figure 2. A binary search tree

Traversing BST

A common requirement when using sets and maps is to go through the elements in order. With binary search trees, traversing from left to right is known as **inordertree traversal**. In a tree where each node has a value and two pointers to the left and right children, inorder tree traversal can be thought of as:

```
Procedure Inorder_traversal(Node n)
    if(n == nil)
        return;
    Inorder_traversal(n.left_subtree);
    Print(n.value);
    Inorder_traversal(n.right_subtree);
…
Inorder_traversal(root);
```

Operations on BST (insertion, deletion and retrieval)

BST insertion:
When adding a new node to a binary search tree, note that the new node will always be a leaf in the tree. To insert a new node, all we will do is navigate the BST starting from the root. If the new node value is smaller than the current node value, we go left – if it is larger, we go right. When we reach a leaf node, the last step is to attach the new node as a child to this leaf node in a way that preserves the BST constraint.

For example, consider we want to add a new node with value 4 to the BST in Figure 1. Here are the steps we will go through:

- Let the current node = root = 5.
- As the new node is smaller than the current node (4 < 5), we will go left and set current node to 2.
- As the new node is greater than current node (4 > 2), we will go right and set the current node to 3.
- We've reached a leaf, so the last step is to attach the new node to the right of the current node. Here is how the new BST looks:

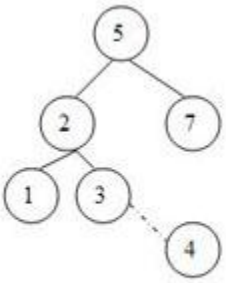


Figure 3. BST after adding node 4

BST Deletion:
Deleting a node from BST is a little more subtle. Formally there are three cases for deleting node n from a BST:

- If n has no children, we only have to remove n from the tree.
- If n has a single child, we remove n and connect its parent to its child.
- If n has two children, we need to :
    - Find the smallest node that is larger than n, call it m.
    - Remove m from the tree (if you have reached this case then m will always have no left child, though I'll leave the proof to the reader), so we apply one or the other of the above cases to do this.
    - Replace the value of n with m.

Figure 4 shows these three cases in action.

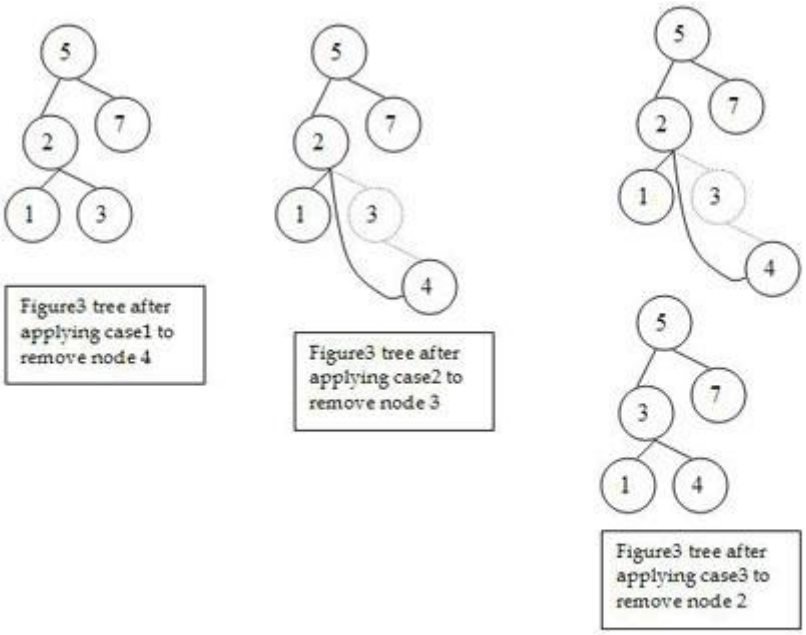


Figure 4. BST deletion

BST Retrieval:
Retrieving an element from binary search trees requires simple navigation, starting from the root and going left, if the current node is larger than the node we are looking for, or going right otherwise.

Any of these primitive operations on BST run in O(h) time, where h is the tree height, so the smaller the tree height the better running time operations will achieve. The problem with BST is that, depending on the order of inserting elements in the tree, the tree shape can vary. In the worst cases (such as inserting elements in order) the tree will look like a linked list in which each node has only a right child. This yields O(n) for primitive operations on the BST, with n the number of nodes in the tree.

To solve this problem many variations of binary search trees exist. Of these variations, red-black trees provide a well-balanced BST that guarantees a logarithmic bound on primitive operations.

Red-black Trees

Red-black trees are an evolution of binary search trees that aim to keep the tree balanced without affecting the complexity of the primitive operations. This is done by coloring each node in the tree with either red or black and preserving a set of properties that guarantee that the deepest path in the tree is not longer than twice the shortest one.

A red-black tree is a binary search tree with the following properties:

1. Every node is colored with either red or black.
2. All leaf (nil) nodes are colored with black; if a node's child is missing then we will assume that it has a nil child in that place and this nil child is always colored black.
3. Both children of a red node must be black nodes.
4. Every path from a node n to a descendent leaf has the same number of black nodes (not counting node n). We call this number the **black height** of n, which is denoted by bh(n).

Figure 5 shows an example of a red-black tree.

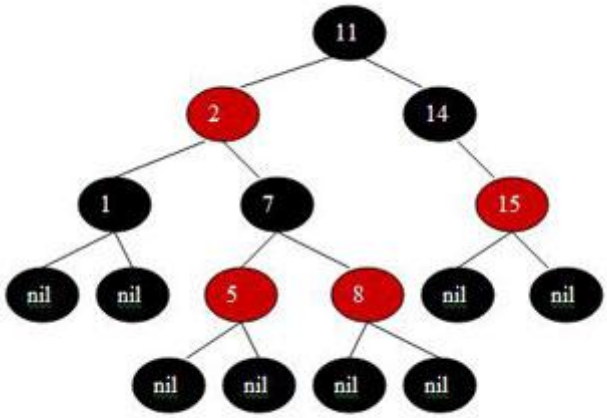


Figure 5. an example of a red-black tree

Using these properties, we can show in two steps that a red-black tree which contains n nodes has a height of O(log n), thus all primitive operations on the tree will be of O(log n) since their order is a function of tree height.

1. First, notice that for a red-black tree with height h, bh(root) is at least h/2 by property 3 above (as each red node strictly requires black children).
2. The next step is to use the following lemma:
    - Lemma: A subtree rooted at node v has at least 2^bh(v) – 1 internal nodes
    - Proof by induction: The basis is when h(v) = 0, which means that v is a leaf node and therefore bh(v) = 0 and the subtree rooted at node v has 2^bh(v)-1 = 2^0-1 = 1-1 = 0 nodes.
    - Inductive hypothesis: if node v1 with height x has 2^bh(v1)-1 internal nodes then node v2 with height x+1 has 2^bh(v2)-1

For any non-leaf node v (height > 0) we can see that the black height of any of its two children is at least equal to bh(v)-1 -- if the child is black, that is, otherwise it is equal to bh(v) . By applying the hypothesis above we conclude that each child has at least 2^[bh(v)-1]-1 internal nodes, accordingly node v has at least
2^[bh(v)-1]-1 + 2^[bh(v)-1]-1 + 1 = 2^bh(v)-1
internal nodes, which ends the proof.

By applying the lemma to the root node (with bh of at least h/2, as shown above) we get
n >= 2^(h/2) – 1
where n is the number of internal nodes of a red-black tree (the subtree rooted at the root). Playing with the equation a little bit we get h <= 2 log (n+1), which guarantees the logarithmic bound of red-black trees.

Rotations

How does inserting or deleting nodes affect a red-black tree? To ensure that its color scheme and properties don't get thrown off, red-black trees employ a key operation known as **rotation**. Rotation is a binary operation, between a parent node and one of its children, that swaps nodes and modifys their pointers while preserving the inorder traversal of the tree (so that elements are still sorted).

There are two types of rotations: left rotation and right rotation. Left rotation swaps the parent node with its right child, while right rotation swaps the parent node with its left child. Here are the steps involved in for left rotation (for right rotations just change "left" to "right" below):

- Assume node x is the parent and node y is a non-leaf right child.
- Let y be the parent and x be its left child.
- Let y's left child be x's right child.

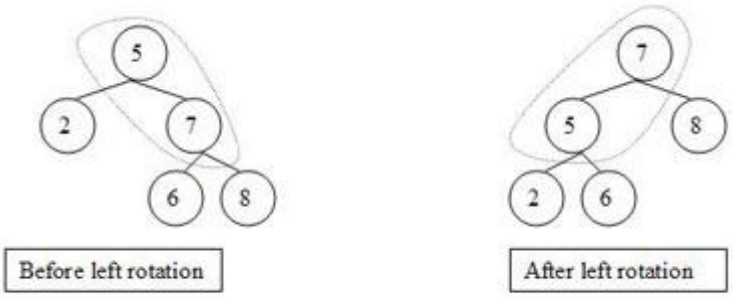


Figure 6. Left rotation

Operations on red-black tree (insertion, deletion and retrieval)

Red-black tree operations are a modified version of BST operations, with the modifications aiming to preserve the properties of red-black trees while keeping the operations complexity a function of tree height.

Red-black tree insertion:
Inserting a node in a red-black tree is a two step process:

1. A BST insertion, which takes O(log n) as shown before.
2. Fixing any violations to red-black tree properties that may occur after applying step 1. This step is O(log n) also, as we start by fixing the newly inserted node, continuing up along the path to the root node and fixing nodes along that path. Fixing a node is done in constant time and involves re-coloring some nodes and doing rotations.

Accordingly the total running time of the insertion process is O(log n). Figure 7 shows the red-black tree in figure 5 before and after insertion of a node with value 4. You can see how the swap operations modified the tree structure to keep it balanced.

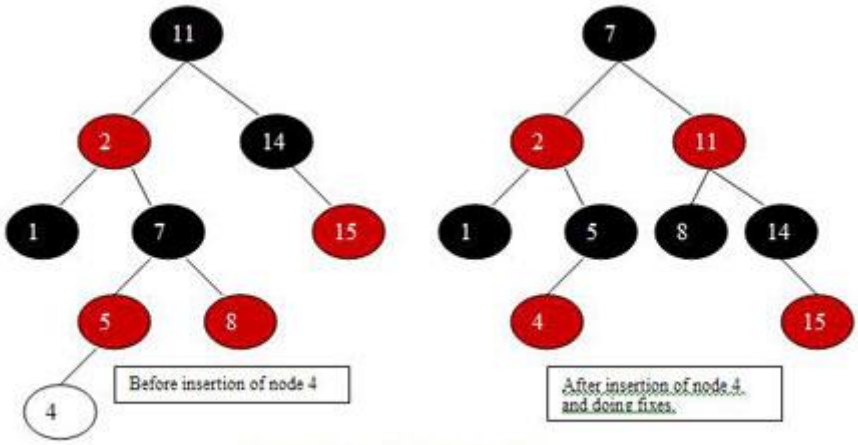


Figure 7. Insertion in red-black tree

Red-black tree deletion:
The same concept behind red-black tree insertions applies here. Removing a node from a red-black tree makes use of the BST deletion procedure and then restores the red-black tree properties in O(log n). The total running time for the deletion process takes O(log n) time, then, which meets the complexity requirements for the primitive operations.

Red-black tree retrieval:
Retrieving a node from a red-black tree doesn't require more than the use of the BST procedure, which takes O(log n) time.

Conclusion

Although you may never need to implement your own set or map classes, thanks to their common built-in support, understanding how these data structures work should help you better assess the performance of your applications and give you more insight into what structure is right for a given task. For more practice with these concepts, check out these problems from the TopCoder archive that involve trees:

- MonomorphicTyper (SRM 286)
- PendingTasks (TCHS SRM 8)
- RedBlack (SRM 155)
- DirectoryTree (SRM 168)
- EncodingTrees (SRM 261)
- AntiChess (SRM 266)
- IncompleteBST (SRM 319)

A previous series of articles covered the basic tools of computational geometry. In this article I'll explore some more advanced algorithms that can be built from these basic tools. They are all based on the simple but powerful idea of a sweep line: a vertical line that is conceptually "swept" across the plane. In practice, of course, we cannot simulate all points in time and so we consider only some discrete points.

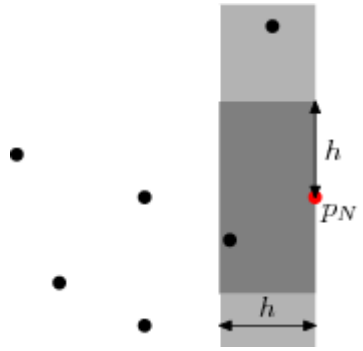
In several places I'll refer to the Euclidean and Manhattan distances. The Euclidean distance is the normal, everyday distance given by Pythagoras' Theorem. The Manhattan distance between points (x_1, y_1) and (x_2, y_2) is the distance that must be travelled while moving only horizontally or vertically, namely $|x_1 - x_2| + |y_1 - y_2|$. It is called the Manhattan distance because the roads in Manhattan are laid out in a grid and so the Manhattan distance is the distance that must be travelled by road (it is also called the "taxicab distance," or more formally the L_1 metric).

In addition, a balanced binary tree is used in some of the algorithms. Generally you can just use a set in C++ or a TreeSet in Java, but in some cases this is insufficient because it is necessary to store extra information in the internal nodes.

Closest pair

Given a set of points, find the pair that is closest (with either metric). Of course, this can be solved in $O(N^2)$ time by considering all the pairs, but a line sweep can reduce this to O(N log N).

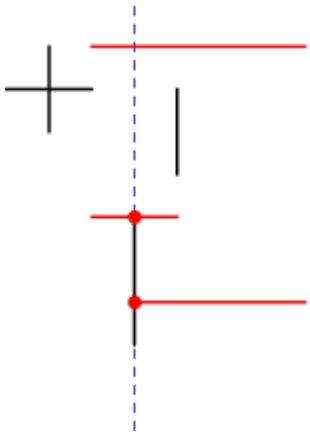
Suppose that we have processed points 1 to N − 1 (ordered by X) and the shortest distance we have found so far is h. We now process point N and try to find a point closer to it than h. We maintain a set of all already-processed points whose X coordinates are within h of point N, as shown in the light grey rectangle. As each point is processed, it is added to the set, and when we move on to the next point or when h is decreased, points are removed from the set. The set is ordered by y coordinate. A balanced binary tree is suitable for this, and accounts for the log N factor.



To search for points closer than h to point N, we need only consider points in the active set, and furthermore we need only consider points whose y coordinates are in the range $y_N - h$ to $y_N + h$ (those in the dark grey rectangle). This range can be extracted from the sorted set in O(log N) time, but more importantly the number of elements is O(1) (the exact maximum will depend on the metric used), because the separation between any two points in the set is at least h. It follows that the search for each point requires O(log N) time, giving a total of O(N log N).

Line segment intersections

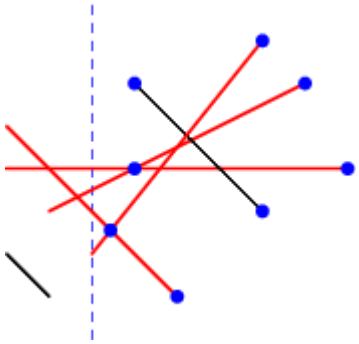
We'll start by considering the problem of returning all intersections in a set of horizontal and vertical line segments. Since horizontal lines don't have a single X coordinate, we have to abandon the idea of sorting objects by X. Instead, we have the idea of an *event*: an X coordinate at which something interesting happens. In this case, the three types of events are: start of a horizontal line, end of a horizontal line, and a vertical line. As the sweep line moves, we'll keep an *active set* of horizontal lines cut by the sweep line, sorted by Y value (the red lines in the figure).



To handle either of the horizontal line events, we simply need to add or remove an element from the set. Again, we can use a balanced binary tree to guarantee O(log N) time for these operations. When we hit a vertical line, a range search immediately gives all the horizontal lines that it cuts. If horizontal or vertical segments can overlap there is some extra work required, and we must also consider whether lines with coincident endpoints are considered to intersect, but none of this affects the computational complexity.

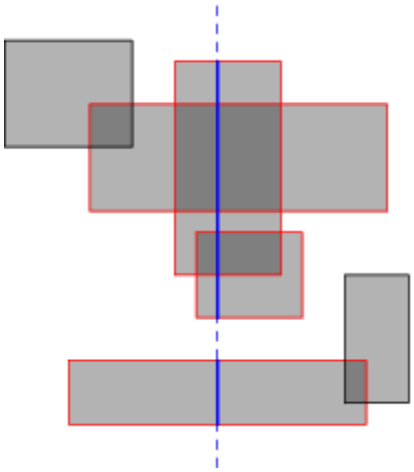
If the intersections themselves are required, this takes O(N log N + I) time for I intersections. By augmenting the binary tree structure (specifically, by storing the size of each sub-tree in the root of that sub-tree), it is possible to count the intersections in O(N log N) time.

In the more general case, lines need not be horizontal or vertical, so lines in the active set can exchange places when they intersect. Instead of having all the events pre-sorted, we have to use a priority queue and dynamically add and remove intersection events. At any point in time, the priority queue contains events for the end-points of line-segments, but also for the intersection points of *adjacent* elements of the active set (providing they are in the future). Since there are O(N + I) events that will be reached, and each requires O(log N) time to update the active set and the priority queue, this algorithm takes O(N log N + I log N) time. The figure below shows the future events in the priority queue (blue dots); note that *not* all future intersections are in the queue, either because one of the lines isn't yet active, or because the two lines are not yet adjacent in the active list.



Area of the union of rectangles

Given a set of axis-aligned rectangles, what is the area of their union? Like the line-intersection problem, we can handle this by dealing with events and active sets. Each rectangle has two events: left edge and right edge. When we cross the left edge, the rectangle is added to the active set. When we cross the right edge, it is removed from the active set.



We now know which rectangles are cut by the sweep line (red in the diagram), but we actually want to know the length of sweep line that is cut (the total length of the solid blue segments). Multiplying this length by the horizontal distance between events gives the area swept out between those two events.

We can determine the cut length by running the same algorithm in an inner loop, but rotated 90 degrees. Ignore the inactive rectangles, and consider a horizontal sweep line that moves top-down. The events are now the horizontal edges of the active rectangles, and every time we cross one, we can simply increment or decrement a counter that says how many rectangles overlap at the current point. The cut length increases as long as the counter is non-zero. Of course, we do not increase it continuously, but rather while moving from one event to the next.

With the right data structures, this can be implemented in $O(N^2)$ time (hint: use a boolean array to store the active set rather than a balanced binary tree, and pre-sort the entire set of horizontal edges). In fact the inner line sweep can be replaced by some clever binary tree

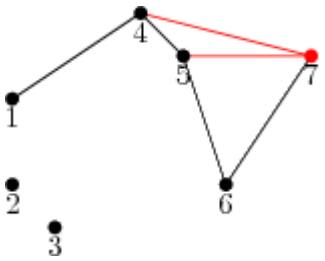
manipulation to reduce the overall time to O(N log N), but that is more a problem in data structures than in geometry, and is left as an exercise for the reader. The algorithm can also be adapted to answer similar questions, such as the total perimeter length of the union or the maximum number of rectangles that overlap at any point.

Convex hull

The *convex hull* of a set of points is the smallest convex polygon that surrounds the entire set, and has a number of practical applications. An efficient method that is often used in contests is the Graham scan [2], which requires a sort by angle. This isn't as easy as it looks at first, since computing the actual angles is expensive and introduces problems with numeric error. A simpler yet equally efficient algorithm is due to Andrew [1], and requires only a sort by X for a line sweep (although Andrew's original paper sorts by Y and has a few optimizations I won't discuss here).

Andrew's algorithm splits the convex hull into two parts, the upper and lower hull. Usually these meet at the ends, but if more than one points has minimal (or maximal) X coordinate, then they are joined by a vertical line segment. We'll describe just how to construct the upper hull; the lower hull can be constructed in similar fashion, and in fact can be built in the same loop.

To build the upper hull, we start with the point with minimal X coordinate, breaking ties by taking the largest Y coordinate. After this, points are added in order of X coordinate (always taking the largest Y value when multiple points have the same X value). Of course, sometimes this will cause the hull to become concave instead of convex:

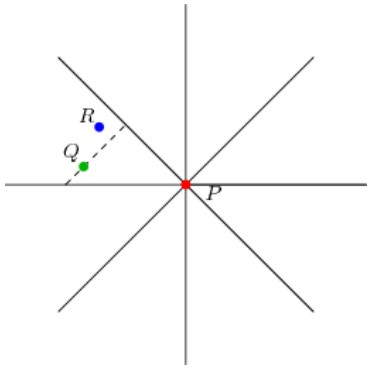


The black path shows the current hull. After adding point 7, we check whether the last triangle (5, 6, 7) is convex. In this case it isn't, so we delete the second-last point, namely 6. The process is repeated until a convex triangle is found. In this case we also examine (4, 5, 7) and delete 5 before examining (1, 4, 7) and finding that it is convex, before proceeding to the next point. This is essentially the same procedure that is used in the Graham scan, but proceeding in order of X coordinate rather than in order of the angle made with the starting point. It may at first appear that this process is $O(N^2)$ because of the inner backtracking loop, but since no point can be deleted more than once it is in fact O(N). The algorithm over-all is O(N log N), because the points must initially be sorted by X coordinate.

Manhattan minimum spanning tree

We can create even more powerful algorithms by combining a line sweep with a divide-and-conquer algorithm. One example is computing the minimum spanning tree of a set of points, where the distance between any pair of points is the Manhattan distance. This is essentially the algorithm presented by Guibas and Stolfi [3].

We first break this down into a simpler problem. Standard MST algorithms for general graphs (e.g., Prim's algorithm) can compute the MST in O((E + N) log N) time for E edges. If we can exploit geometric properties to reduce the number of edges to O(N), then this is merely O(N log N). In fact we can consider, for each point P, only its nearest neighbors in each of the 8 octants of the plane (see the figure below). The figure shows the situation in just one of the octants, the West-Northwest one. Q is the closest neighbour (with the dashed line indicating points at the same Manhattan distance as Q), and R is some other point in the octant. If PR is an edge in a spanning tree, then it can be removed and replaced by either PQ or QR to produce a better spanning tree, because the shape of the octant guarantees that $|QR| \leq |PR|$. Thus, we do not need to consider PR when building the spanning tree.



This reduces the problem to that of finding the nearest neighbour in each octant. We'll just consider the octant shown; the others are no different and can be handled by symmetry. It should be clear that within this octant, finding the nearest neighbour is equivalent to just finding the point with the largest value of x − y, subject to an upper bound on x + y and a lower bound on y, and this is the form in which we'll consider the problem.

Now imagine for the moment that the lower bound on y did not exist. In this case we could solve the problem for every P quite easily: sweep through the points in increasing order of x + y, and Q will be the point with the largest x − y value of those seen so far. This is where the divide-and-conquer principle comes into play: we partition the point set into two halves with a horizontal line, and recursively solve the problem for each half. For points P in the upper half, nothing further needs to be done, because points in the bottom half cannot play Q to their P. For the bottom half, we have to consider that by ignoring the upper half so far we may have missed some closer points. However, we can take these points into account in a similar manner as before: walk through all the points in x + y order, keeping track of the best point in the top half (largest x − y value), and for each point in the bottom half, checking whether this best top-half point is better than the current neighbour.

So far I have blithely assumed that any set of points can be efficiently partitioned on Y and also walked in x + y order without saying how this should be done. In fact, one of the most beautiful aspects of this class of divide-and-conquer plus line-sweep algorithms is that it has essentially the same structure as a merge sort, to the point that a merge-sort by x + y can be folded into the algorithm in such a way that each subset is sorted on x + y just when this is needed (the points initially all being sorted on Y). This gives the algorithm a running time of O(N log N).

The idea of finding the closest point within an angle range can also be used to solve the Euclidean MST problem, but the O(N log N) running time is no longer guaranteed in the worst cases, because the distance is no longer a linear equation. It is actually possible to compute the Euclidean MST in O(N log N) time, because it is a subset of the Delaunay triangulation.

Sample problems

BoxUnion
: This is the union of area of rectangles problem above. In this instance there are at most three rectangles which makes simpler solutions feasible, but you can still use this to practice.

CultureGrowth
: While written in a misleading fashion, the task is just to compute the area of the convex hull of a set of points.

PowerSupply
: For each power line orientation, sweep the power line in the perpendicular direction. Consumers are added D units ahead of the sweep and dropped D units behind the sweep. In fact, the low constraints mean that the connected set can be computed from scratch for each event.

ConvexPolygons
: The events of interest are the vertices of the two polygons, and the intersection points of their edges. Between consecutive events, the section cut by the sweep line varies linearly. Thus, we can sample the cut area at the mid-point X value of each of these regions to get the average for the whole region. Sampling at these mid-points also eliminates a lot of special-case handling, because the sweep line is guaranteed not to pass anywhere near a vertex. Unlike the solution proposed in the match editorial, the only geometric tool required is line-line intersection.

Conclusion

Like dynamic programming, the sweep line is an extremely powerful tool in an algorithm competitor's toolkit because it is not simply an algorithm: it is an algorithm pattern that can be tailored to solve a wide variety of problems, including other textbooks problems that I have not discussed here (Delaunay triangulations, for example), but also novel problems that may have been created specifically for a contest. In TopCoder the small constraints often mean that one can take shortcuts (such as processing each event from scratch rather than incrementally, and in arbitrary order), but the concept of the sweep line is still useful in finding a solution.

This article covers the so-called "min-cost flow" problem, which has many applications for both TopCoder competitors and professional programmers. The article is targeted to readers who are not familiar with the subject, with a focus more on providing a general understanding of the ideas involved rather than heavy theory or technical details; for a more in-depth look at this topic, check out the references at the end of this article, in particular [1]. In addition, readers of this article should be familiar with the basic concepts of graph theory -- including shortest paths [4], paths with negative cost arcs, negative cycles [1] -- and maximum flow theory's basic algorithms [3].

The article is divided into three parts. In Part 1, we'll look at the problem itself. The next part will describe three basic algorithms, and Part 3 some applications to the problem will be covered in Part 3.

Statement of the Problem

What is the minimum cost flow problem? Let's begin with some important terminology.

Let $G = (V, E)$ be a directed network defined by a set V of vertexes (nodes) and set E of edges (arcs). For each edge $(i, j) \in E$ we associate a **capacity** u_{ij} that denotes the maximum amount that can flow on the edge. Each edge $(i, j) \in E$ also has an associated **cost** c_{ij} that denotes the cost per unit flow on that edge.

We associate with each vertex $i \in V$ a number b_i. This value represents supply/demand of the vertex. If $b_i > 0$, node i is a **supply node**; if $b_i < 0$, node i is a **demand node** (its demand is equal to $-b_i$). We call vertex i a **transshipment** if b_i is zero.

For simplification, let's call G a **transportation network** and write $G = (V, E, u, c, b)$ in case we want to show all the network parameters explicitly.

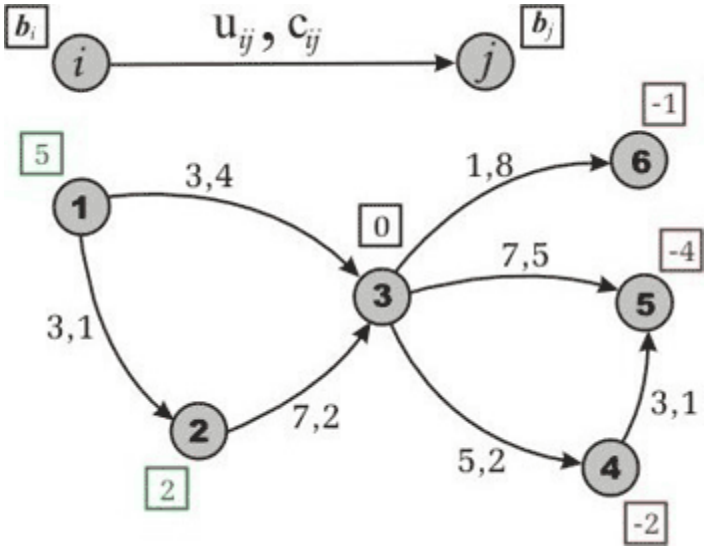


Figure 1. An example of the transportation network. In this we have 2 supply vertexes (with supply values 5 and 2), 3 demand vertexes (with demand values 1, 4 and 2), and 1 transshipment node. Each edge has two numbers, capacity and cost, divided by comma.

Representing the flow on arc $(i, j) \in E$ by x_{ij}, we can obtain the optimization model for the minimum cost flow problem:

$$\text{Minimize } z(x) = \sum_{(i,j) \in E} c_{ij} x_{ij}$$

subject to

$$\sum_{\{j:(i,j)\in E\}} x_{ij} - \sum_{\{j:(j,i)\in E\}} x_{ji} = b_i \quad \text{for all } i \in V,$$

$$0 \le x_{ij} \le u_{ij} \quad \text{for all } (i,j) \in E.$$

The first constraint states that the total outflow of a node minus the total inflow of the node must be equal to mass balance (supply/demand value) of this node. This is known as the **mass balance constraints**. Next, the **flow bound constraints** model physical capacities or restrictions imposed on the flow's range. As you can see, this optimization model describes a typical relationship between warehouses and shops, for example, in a case where we have only one kind of product. We need to satisfy the demand of each shop by transferring goods from the subset of warehouses, while minimizing the expenses on transportation.

This problem could be solved using simplex-method, but in this article we concentrate on some other ideas related to network flow theory. Before we move on to the three basic algorithms used to solve the minimum cost flow problem, let's review the necessary theoretical base.

Finding a solution

When does the minimum cost flow problem have a feasible (though not necessarily optimal) solution? How do we determine whether it is possible to translate the goods or not?

If $\delta \stackrel{def}{=} \sum_{i \in V} b_i \neq 0$ then the problem has no solution, because either the supply or the demand dominates in the network and the mass balance constraints come into play.

We can easily avoid this situation, however, if we add a special node r with the supply/demand value $b_r = -\delta$. Now we have two options: If $\delta > 0$ (supply dominates) then for each node $i \in V$ with $b_i > 0$ we add an arc (i,r) with infinite capacity and zero cost; otherwise (demand dominates), for each node $i \in V$ with $b_i < 0$, we add an arc (r,i) with the same properties. Now we have a new network with $\sum_{i \in V \cup \{r\}} b_i = 0$ -- and it is easy to prove that this network has the same optimal value as the objective function.

Consider the vertex r as a rubbish or scrap dump. If the shops demand is less than what the warehouse supplies, then we have to eject the useless goods as rubbish. Otherwise, we take the missing goods from the dump. This would be considered shady in real life, of course, but for our purposes it is very convenient. Keep in mind that, in this case, we cannot say what exactly the "solution" of the corrected (with scrap) problem is. And it is up to the reader how to classify the emergency uses of the "dump." For example, we can suggest that goods remain in the warehouses or some of the shop's demands remain unsatisfied.

Even if we have $\delta = 0$ we are not sure that the edge's capacities allow us to transfer enough flow from supply vertexes to demand ones. To determine if the network has a feasible flow, we want to find any transfer way what will satisfy all the problem's constraints. Of course, this feasible solution is not necessarily optimal, but if it is absent we cannot solve the problem.

Let us introduce a source node s and a sink node t. For each node $i \in V$ with $b_i > 0$, we add a source arc (s,i) to G with capacity b_i and cost 0. For each node $i \in V$ with $b_i < 0$, we add a sink arc (i,t) to G with capacity $-b_i$ and cost 0.

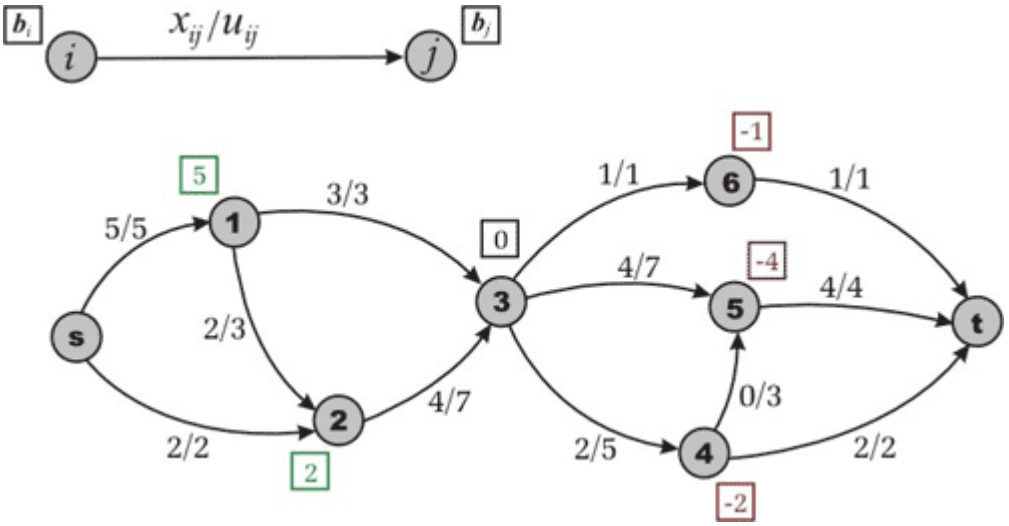


Figure 2. Maximum flow in the transformed network. For simplicity we are ignoring the costs.

The new network is called a **transformed network**. Next, we solve a maximum flow problem from s to t (ignoring costs, see fig.2). If the maximum flow saturates all the source and sink arcs, then the problem has a feasible solution; otherwise, it is infeasible. As for why this approach works, we'll leave its proof to the reader.

Having found a maximum flow, we can now remove source, sink, and all adjacent arcs and obtain a feasible flow in G. How do we detect whether the flow is optimal or not? Does this flow minimize costs of the objective function z? We usually verify "optimality conditions" for the answer to these questions, but let us put them on hold for a moment and discuss some assumptions.

Now, suppose that we have a network that has a feasible solution. Does it have an optimal solution? If our network contains the negative cost cycle of infinite capacity then the objective function will be unbounded. However, in some tasks, we are able to assign finite capacity to each uncapacitated edge escaping such a situation.

So, from the theoretical point of view, for any minimum cost flow problem we have to check some conditions: The supply/demand balance, the existence of a feasible solution, and the last situation with uncapacitated negative cycles. These are necessary conditions for resolving the problem. But from the practical point of view, we can check the conditions while the solution is being found.

Assumptions

In understanding the basics of network flow theory it helps to make some assumptions, although sometimes they can lead to a loss of generality. Of course, we could solve the problems without these assumptions, but the solutions would rapidly become too complex. Fortunately, these assumptions are not as restrictive as they might seem.

Assumption 1. *All data (u_{ij}, c_{ij}, b_i) are integral.*

As we have to deal with a computer, which works with rational numbers, this assumption is not restrictive in practice. We can convert rational numbers to integers by multiplying by a suitable large number.

Assumption 2. *The network is directed.*

If the network were undirected we would transform it into a directed one. Unfortunately, this transformation requires the edge's cost to be nonnegative. Let's validate this assumption.

To transform an undirected case to a directed one, we replace each undirected edge connecting vertexes i and j by two directed edges (i, j) and (j, i), both with the capacity and cost of the replaced arc. To establish the correctness of this transformation, first we note that for undirected arc $(i, j) \in E$ we have constraint $x_{ij} + x_{ji} \le u_{ij}$ and the term $c_{ij}x_{ij} + c_{ij}x_{ji}$ in the objective function. Given that $c_{ij} \ge 0$ we see that in some optimal solution either x_{ij} or x_{ji} will be zero. We call such a solution non-overlapping. Now it is easy to make sure (and we leave it to the reader) that every non-overlapping flow in the original network has an associated flow in the transformed network with the same cost, and vise versa.

Assumption 3. *All costs associated with edges are nonnegative.*

This assumption imposes a loss of generality. We will show below that if a network with negative costs had no negative cycle it would be possible to transform it into one with nonnegative costs. However, one of the algorithms (namely cycle-canceling algorithm) which we are going to discuss is able to work without this assumption.

For each vertex $i \in V$ let's denote by π_i a number associated with the vertex and call it the **potential** of node i. Next define the **reduced cost** c_{ij}^{π} of an edge $(i, j) \in E$ as

$$c_{ij}^{\pi} = c_{ij} + \pi_i - \pi_j$$

How does our objective value change? Let's denote reduced value by $z(x,\pi)$. Evidently, if $\pi = 0$, then

$$z(x,0) = \sum_{(i,j) \in E} c_{ij} x_{ij} = z(x)$$

For other values of π we obtain following result:

$$z(x,\pi) = \sum_{(i,j) \in E} c_{ij}^{\pi} x_{ij} = \sum_{(i,j) \in E} (c_{ij} + \pi_i - \pi_j) x_{ij} =$$

$$z(x) + \sum_{(i,j) \in E} \pi_i x_{ij} - \sum_{(i,j) \in E} \pi_j x_{ij} =$$

$$= z(x) + \sum_{i \in V} \pi_i \sum_{\{j:(i,j) \in E\}} x_{ij} - \sum_{j \in V} \pi_j \sum_{\{i:(i,j) \in E\}} x_{ij} =$$

$$= z(x) + \sum_{i \in V} \pi_i \left(\sum_{\{j:(i,j) \in E\}} x_{ij} - \sum_{\{j:(j,i) \in E\}} x_{ji} \right) = z(x) + \sum_{i \in V} \pi_i b_i$$

For a fixed π, the difference $z(x,\pi) - z(x)$ is constant. Therefore, a flow that minimizes $z(x,\pi)$ also minimizes z(x) and vice versa. We have proved:

Theorem 1. *For any node potential π the minimum cost flow problems with edge costs c_{ij} or c_{ij}^{π} have the same optimal solutions. Moreover,*

$$z(x,\pi) = z(x) + \sum_{i \in V} \pi_i b_i$$

The following result contains very useful properties of reduced costs.

Theorem 2. *Let G be a transportation network. Suppose P is a directed path from $a \in V$ to $b \in V$. Then for any node potential π*

$$\sum_{(i,j) \in P} c_{ij}^{\pi} = \sum_{(i,j) \in P} c_{ij} + \pi_a - \pi_b$$

Suppose W is a directed cycle. Then for any node potential π

$$\sum_{(i,j) \in W} c_{ij}^{\pi} = \sum_{(i,j) \in W} c_{ij}$$

This theorem implies the following reasoning. Let's introduce a vertex s and for each node $i \in V$, we add an arc (s,i) to G with some positive capacity and zero cost. Suppose that for each $i \in V$ number π_i denotes length of the shortest path from s to i with respect to cost function c.

(Reminder: there is no negative length cycle). If so, one can justify (or read it in [2]) that for each $(i, j) \in E$ the shortest path optimality condition is satisfied:

$$\pi_j \le \pi_i + c_{ij}$$

Since, $c_{ij}^{\pi} = c_{ij} + \pi_i - \pi_j$ and $0 \le \pi_i - \pi_j + c_{ij}$ yields $c_{ij}^{\pi} \ge 0$. Moreover, applying theorem 2, we can note that if G contains negative cycle, it will be negative for any node potential π in reduced network. So, if the transportation network has no negative cycle, we will be able to reduce costs and make them positive by finding the shortest paths from the introduced vertex s, otherwise, our assumption doesn't work. If the reader asks how to find the shortest path in graph with negative costs, I'll refer you back to the basics of the graph theory. One can use Bellman-Ford (label-correcting) algorithm to achieve this goal [1, 2].

Remember this reduced cost technique, since it appears in many applications and other algorithms (for example, Johnson's algorithm for all pair shortest path in sparse networks uses it [2]).

Assumption 4. *The supply/demand at the vertexes satisfy the condition* $\sum_{i \in V} b_i = 0$ *and the minimum cost flow problem has a feasible solution.*

This assumption is a consequence of the "Finding a Solution" section of this article. If the network doesn't satisfy the first part of this assumption, we can either say that the problem has no solution or make corresponding transformation according to the steps outlined in that section. If the second part of the assumption isn't met then the solution doesn't exist.

By making these assumptions we do transform our original transportation network. However, many problems are often given in such a way which satisfies all the assumptions.

Working with Residual Networks

Let's consider the concept of residual networks from the perspective of min-cost flow theory. You should be familiar with this concept thanks to maximum flow theory, so we'll just extend it to minimum cost flow theory.

We start with the following intuitive idea. Let G be a network and x be a feasible solution of the minimum cost flow problem. Suppose that an edge (i,j) in E carries x_{ij} units of flow. We define the residual capacity of the edge (i,j) as $r_{ij} = u_{ij} - x_{ij}$. This means that we can send an additional r_{ij} units of flow from vertex i to vertex j. We can also cancel the existing flow x_{ij} on the arc if we send up x_{ij} units of flow from j to i over the arc (i,j). Now note that sending a unit of flow from i to j along the arc (i,j) increases the objective function by c_{ij}, while sending a unit of flow from j to i on the same arc decreases the flow cost by c_{ij}.

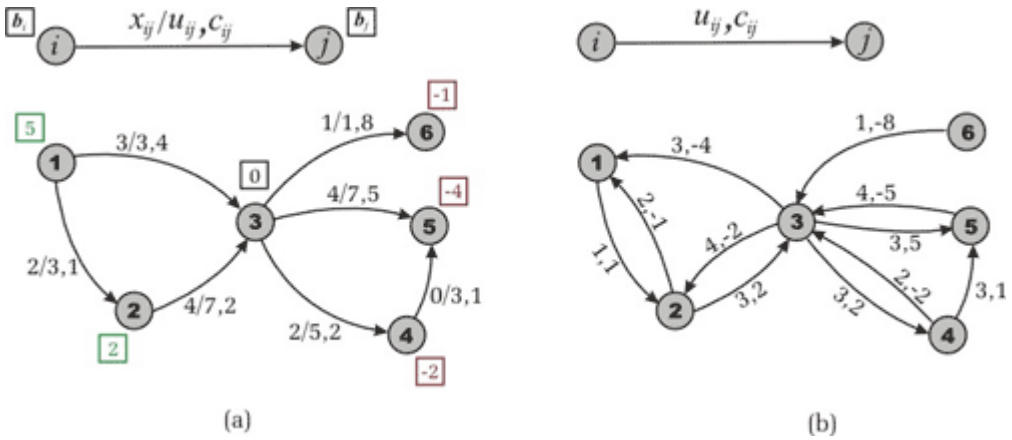


Figure 1. The transportation network from Part 1. (a) A feasible solution. (b) The residual network with respect to the found feasible solution.

Based on these ideas we define the residual network with respect to the given flow x as follows. Suppose we have a transportation network G = (V,E). A feasible solution x engenders a new (residual) transportation network, which we are used to defining by $G_x = (V,E_x)$, where E_x is a set of residual edges corresponding to the feasible solution x.

What is E_x? We replace each arc (i,j) in E by two arcs (i,j), (j,i): the arc (i,j) has cost c_{ij} and (residual) capacity $r_{ij} = u_{ij} - x_{ij}$, and the arc (j,i) has cost $-c_{ij}$ and (residual) capacity $r_{ji}=x_{ij}$. Then we construct the set E_x from the new edges with a positive residual capacity. Look at Figure 1 to make sure that you understand the construction of the residual network.

You can notice immediately that such a definition of the residual network has some technical difficulties. Let's sum them up:

- If G contains both the edges (i,j) and (j,i) (remember assumption 2) the residual network may contain four edges between i and j (two parallel arcs from i to j and two contrary). To avoid this situation we have two options. First, transform the original network to one in which the network contains either edge (i,j) or edge (j,i), but not both, by splitting the vertexes i and j. Second, represent our network by the adjacency list, which is handling parallel arcs. We could even use two adjacency matrixes if it were more convenient.
- Let's imagine now that we have a lot of parallel edges from i to j with different costs. Unfortunately, we can't merge them by summarizing their capacities, as we could do while we were finding the maximum flow. So, we need to keep each of the parallel edges in our data structure separate.

The proof of the fact that there is a one-to-one correspondence between the original and residual networks is out the scope of this article, but you could prove all the necessary theorems as it was done within the maximum flow theory, or by reading [1].

Cycle-canceling Algorithm

This section describes the negative cycle optimality conditions and, as a consequence, cycle-canceling algorithm. We are starting with this important theorem:

Theorem 1 (Solution Existence). *Let G be a transportation network. Suppose that G contains no uncapacitated negative cost cycle and there exists a feasible solution of the minimum cost flow problem. Then the optimal solution exists.*

Proof. One can see that the minimum cost flow problem is a special case of the linear programming problem. The latter is well known to have an optimal solution if it has a feasible solution and its objective function is bounded. Evidently, if G doesn't contain an uncapacitated negative cycle then the objective function of the minimum cost flow problem is bounded from below -- therefore, the assertion of the theorem follows forthwith.

We will use the following theorem without proof, because we don't want our article to be overloaded with difficult theory, but you can read the proof in [1].

Theorem 2 (Negative Cycle Optimality Conditions). *Let x* be a feasible solution of a minimum cost flow problem. Then x* is an optimal solution if and only if the residual network G_{x^*} contains no negative cost (directed) cycle.*

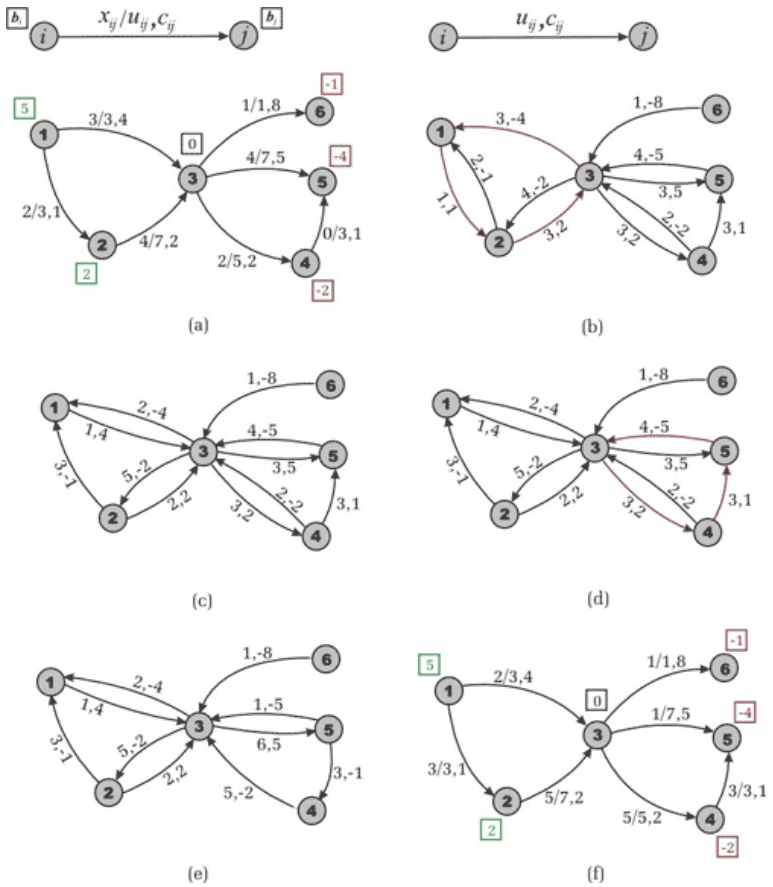


Figure 2. Cycle-Canceling Algorithm, example of the network from Figure 1. (a) We have a feasible solution of cost 54. (b) A negative cycle 1-2-3-1 is detected in the residual network. Its cost is -1 and capacity is 1. (c) The residual network after augmentation along the cycle. (d) Another

negative cost cycle 3-4-5-3 is detected. It has cost -2 and capacity 3. (e) The residual network after augmentation. It doesn't contain negative cycles. (f) Optimal flow cost value is equal to 47.

This theorem gives the cycle-canceling algorithm for solving the minimum cost flow problem. First, we use any maximum flow algorithm [3] to establish a feasible flow in the network (remember assumption 4). Then the algorithm attempts to improve the objective function by finding negative cost cycles in the residual network and augmenting the flow on these cycles. Let us specify a program in pseudo code like it is done in [1].

Cycle-Canceling

1 Establish a feasible flow x in the network
2 **while** (G_x contains a negative cycle) **do**
3 identify a negative cycle W
4 $\delta \leftarrow \min\{r_{ij} : (i,j) \in W\}$
5 augment δ units of flow along the cycle W
6 update G_x

How many iterations does the algorithm perform? First, note that due to assumption 1 all the data is integral. After line 1 of the program we have an integral feasible solution x. It implies the integrality of G_x. In each iteration of the cycle in line 2 the algorithm finds the minimum residual capacity in the found negative cycle. In the first iteration δ will be an integer. Therefore, the modified residual capacities will be integers, too. And in all subsequent iterations the residual capacities will be integers again. This reasoning implies:

Theorem 3 (Integrality Property). *If all edge capacities and supplies/demands on vertexes are integers, then the minimum cost flow problem always has an integer solution.*

The cycle-canceling algorithm works in cases when the minimum cost flow problem has an optimal solution and all the data is integral and we don't need any other assumptions.

Now let us denote the maximum capacity of an arc by U and its maximum absolute value of cost by C. Suppose that m denotes the number of edges in G and n denotes the number of vertexes. For a minimum cost flow problem, the absolute value of the objective function is bounded by mCU. Any cycle canceling decreases the objective function by a strictly positive amount. Since we are assuming that all data is integral, the algorithm terminates within O(mCU) iterations. One can use O(nm) algorithm for identifying a negative cycle (for instance, Bellman-Ford's algorithm or label correcting algorithm [1]), and obtain complexity $O(nm^2CU)$ of the algorithm.

Successive Shortest Path Algorithm
The previous algorithm solves the maximum flow problem as a subtask. The successive shortest path algorithm searches for the maximum flow and optimizes the objective function simultaneously. It solves the so-called max-flow-min-cost problem by using the following idea.

Suppose we have a transportation network G and we have to find an optimal flow across it. As it is described in the "Finding a Solution" section we transform the network by adding two vertexes s and t (source and sink) and some edges as follows. For each node i in V with $b_i > 0$, we add a source arc (s,i) with capacity b_i and cost 0. For each node i in V with $b_i < 0$, we add a sink arc (i,t) with capacity $-b_i$ and cost 0.

Then, instead of searching for the maximum flow as usual, we send flow from s to t along the shortest path (with respect to arc costs). Next we update the residual network, find another shortest path and augment the flow again, etc. The algorithm terminates when the residual network contains no path from s to t (the flow is maximal). Since the flow is maximal, it corresponds to a feasible solution of the original minimum cost flow problem. Moreover, this solution will be optimal (and we are going to explain why).

The successive shortest path algorithm can be used when G contains no negative cost cycles. Otherwise, we cannot say exactly what "the shortest path" means. Now let us justify the successive shortest path approach. When the current flow has zero value, the transportation network G doesn't contain a negative cost cycle (by hypothesis). Suppose that after some augmenting steps we have flow x and G_x still contains no negative cycles. If x is maximal then it is optimal, according to theorem 2. Otherwise, let us denote the next successfully found shortest path in G_x by P.

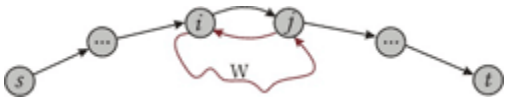


Figure 3. How could a negative cycle appear in a residual network?

Suppose that after augmenting the current flow x along path P a negative cost cycle W turned up in the residual network. Before augmenting there were no negative cycles. This means that there was an edge (i,j) in P (or subpath (i,…,j) in P) the reversal of which (j,i) closed cycle W after the augmentation. Evidently, we could choose another path from s to t, which goes from s to i then from i to j along edges of W then from j to t. Moreover, the cost of this path is less than the cost of P. We have a contradiction to the supposition that P is the shortest.

What do we have? After the last step we have a feasible solution and the residual network contains no negative cycle. The latter is the criterion of optimality.

A simple analysis shows that the algorithm performs at most O(nB) augmentations, where B is assigned to an upper bound on the largest supply of any node. Really, each augmentation strictly decreases the residual capacity of a source arc (which is equal to the supply of the corresponding node). Thanks to the integrality property it decreases by at least one unit. By using an O(nm) algorithm for finding a shortest path (there may be negative edges), we achieve an $O(n^2mB)$ complexity of the successive shortest path algorithm.

Successive Shortest Path
1 Transform network G by adding source and sink
2 Initial flow x is zero
3 **while** (G_x contains a path from s to t) **do**
4 Find any shortest path P from s to t
5 Augment current flow x along P
6 update G_x

Let us reveal the meaning of node potentials from assumption 3. As it is said within assumption 3, we are able to make all edge costs nonnegative by using, for instance, Bellman-Ford's algorithm. Since working with residual costs doesn't change shortest paths (by theorem 2, part 1) we can work with the transformed network and use Dijkstra's algorithm to find the successive shortest path more efficiently. However, we need to keep the edge costs nonnegative on each iteration -- for this purpose, we update node potentials and reduce costs right after the shortest path has been found. The reduce cost function could be written in the following manner:

Reduce Cost (π)
1 **For each** (i,j) in E_x **do**
2 $c_{ij} \leftarrow c_{ij} + \pi_i - \pi_j$
3 $c_{rev(i,j)} \leftarrow 0$

Having found the successive shortest path we need to update node potentials. For each i in V the potential π_i is equal to the length of the shortest paths from s to t. After having reduced the cost of each arc, we will see that along the shortest path from s to i arcs will have zero cost while the arcs which lie out of any shortest path to any vertex will have a positive cost. That is why we assign zero cost to any reversal arc ($c_{rev(i,j)}$) in the Reduce Cost Procedure in line 3. The augmentation (along the found path) adds reversal arc (j,i) and due to the fact that (reduced) cost $c_{ij} = 0$ we make ($c_{rev(i,j)}$) = 0 beforehand.

Why have we denoted cost of reversal arc by ($c_{rev(i,j)}$) instead of c_{ji}? Because the network may contain both arcs (i,j) and (j,i) (remember assumption 2 and "Working with Residual Networks" section). For other arcs (which lie out of the augmenting path) this forcible assignment does nothing, because its reversal arcs will not appear in the residual network. Now we propose a pseudo-code program:

Successive Shortest Path with potentials
1 Transform network G by adding source and sink
2 Initial flow x is zero
3 Use Bellman-Ford's algorithm to establish potentials π
4 Reduce Cost (π)
5 **while** (G_x contains a path from s to t) **do**
6 Find any shortest path P from s to t
7 Reduce Cost (π)
8 Augment current flow x along P
9 update G_x

Before starting the cycle in line 5 we calculate node potentials π and obtain all costs to be nonnegative. We use the same massif of costs c when reducing. In line 6 we use Dijkstra's algorithm to establish a shortest path with respect to the reduced costs. Then we reduce costs and augment flow along the path. After the augmentation all costs will remain nonnegative and in the next iteration Dijkstra's algorithm will work correctly.

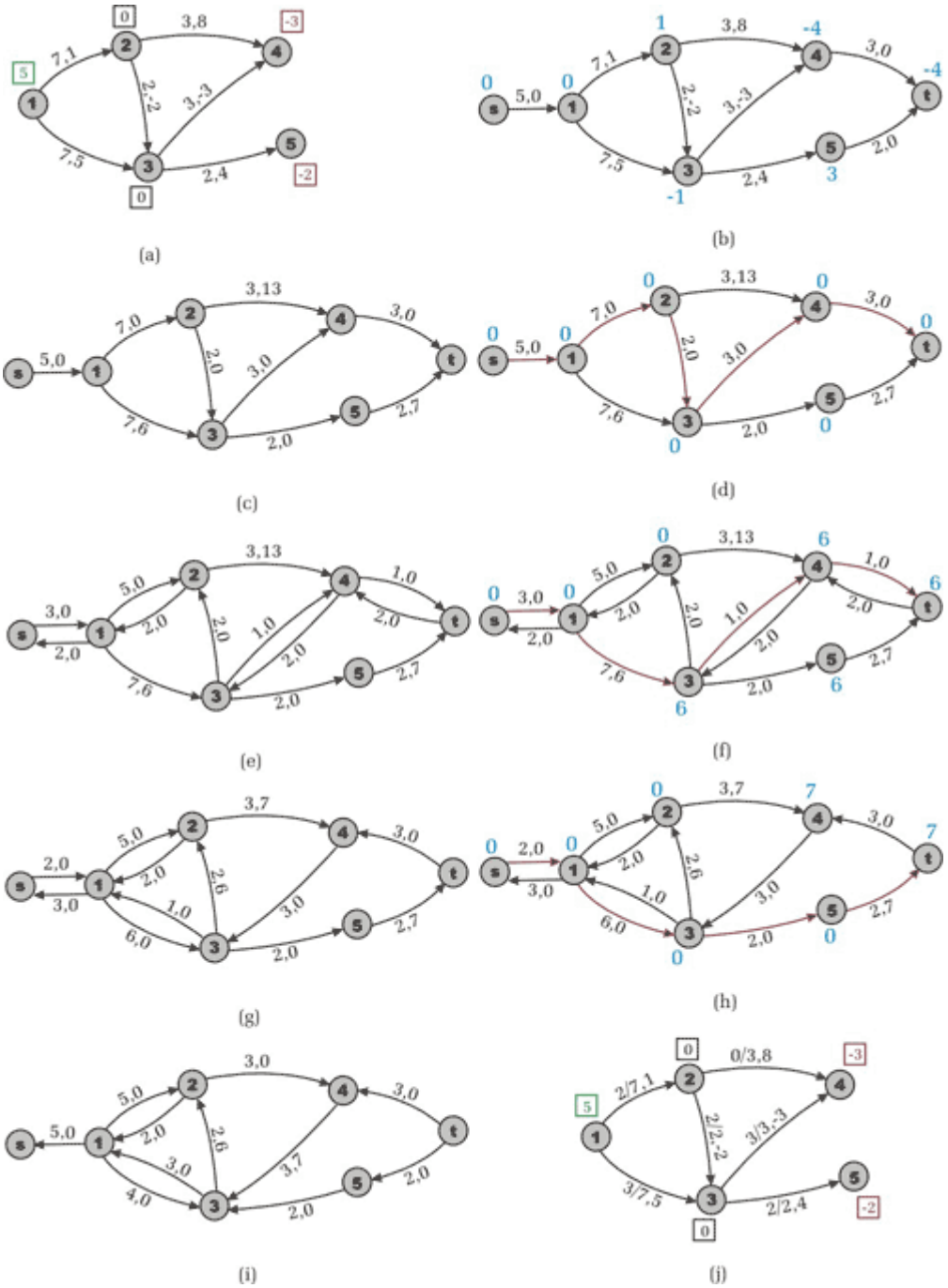


Figure 4. The Successive shortest Path Algorithm. (a) Initial task. (b) Node potentials are calculated after line 3 of the program. (c) Reduced costs after line 4. (d) The first augmenting path s-1-2-3-4-t of capacity 2 is found and new node potentials are calculated. (e) The residual network with reduced costs. (f) The second augmenting path s-1-3-4-t of capacity 1 is found. (g) The residual network with reduced costs. (h) The third shortest augmenting path s-1-3-5-t and new node potentials are found. (i) The residual network contains no augmenting paths. (j) The reconstructed transportation network. Optimal flow has cost 12.

We use Bellman-Ford's algorithm only once to avoid negative costs on edges. It takes O(nm) time. Then O(nB) times we use Dijkstra algorithm, which takes either $O(n^2)$ (simple realization) or O(mlogn) (heap realization for sparse network, [4]) time. Summing up, we receive $O(n^3B)$ estimate working time for simple realization and O(nmBlogn) if using heap. One could even use Fibonacci Heaps to obtain O(nlogn+m) complexity of Dijkstra's shortest path algorithm; however I wouldn't recommend doing so because this case works badly in practice.

Primal-Dual Algorithm

The primal-dual algorithm for the minimum cost flow problem is similar to the successive shortest path algorithm in the sense that it also uses node potentials and shortest path algorithm to calculate them. Instead of augmenting the flow along one shortest path, however, this algorithm increases flow along all the shortest paths at once. For this purpose in each step it uses any maximum flow algorithm to find the

maximum flow through the so called **admissible network**, which contains only those arcs in G_x with a zero reduced cost. We represent the admissible residual network with respect to flow x as G_x°. Let's explain the idea by using a pseudo-code program.

```
Primal-Dual
1  Transform network G by adding source and sink
2  Initial flow x is zero
3  Use Bellman-Ford's algorithm to establish potentials π
4  Reduce Cost ( π )
5  while ( G_x contains a path from s to t ) do
6     Calculate node potential π using Dijkstra's algorithm
7     Reduce Cost ( π )
8     Establish a maximum flow y from s to t in G_x°
9     x ← x + y
10    update G_x
```

For a better illustration look at Figure 5.

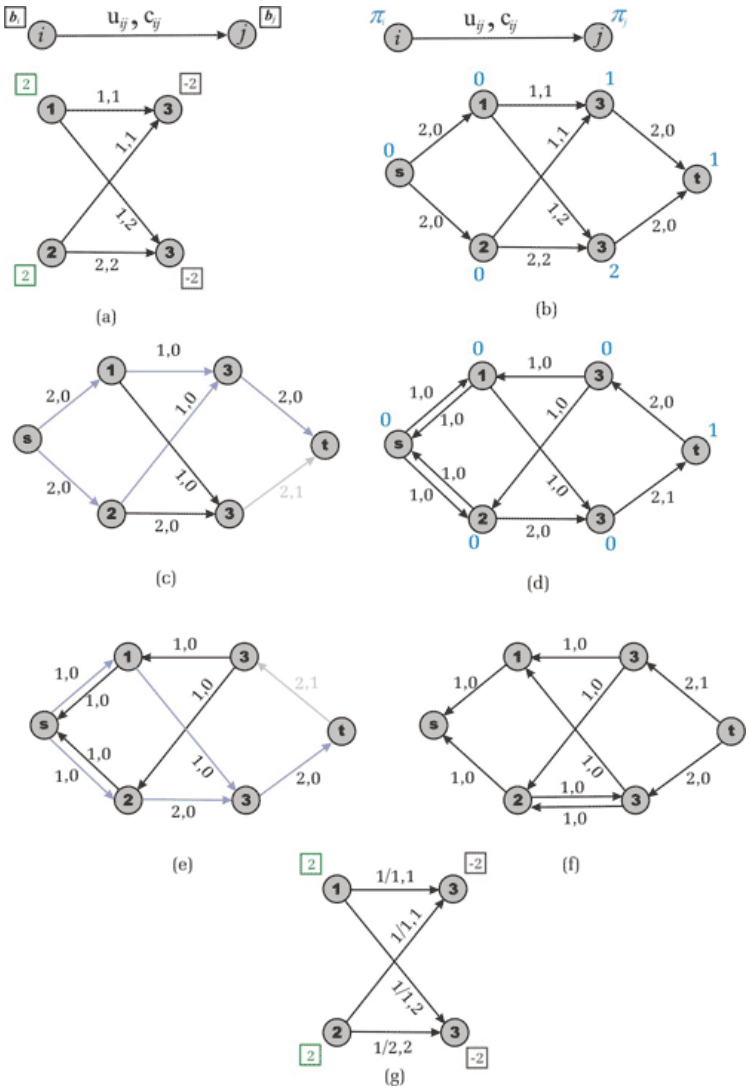


Figure 5. Primal-Dual algorithm. (a) Example network. (b) Node potentials are calculated. (c) The maximum flow in the admissible network. (d) Residual network and new node potentials. (e) The maximum flow in the admissible network. (f) Residual network with no augmenting paths. (g) The optimal solution.

As mentioned above, the primal-dual algorithm sends flow along all shortest paths at once; therefore, proof of correctness is similar to the successive shortest path one.

First, the primal-dual algorithm guarantees that the number of iterations doesn't exceed O(nB) as well as the successive shortest path algorithm. Moreover, since we established a maximum flow in G°_{x}, the residual network G_x contains no directed path from vertex s to vertex t consisting entirely of arcs of zero costs. Consequently, the distance between s and t increases by at least one unit. These observations give a bound of min{nB,nC} on the number of iterations which the primal-dual algorithm performs. Keep in mind, though, that the algorithm incurs the additional expense of solving a maximum flow problem at every iteration. However, in practice both the successive shortest path and the primal-dual algorithm work fast enough within the constraint of 50 vertexes and reasonable supply/demand values and costs.

The Assignment Problem

There are a number of *agents* and a number of *tasks*. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. We have to get all tasks performed by assigning exactly one agent to each task in such a way that the total cost of the assignment is minimal with respect to all such assignments.

In other words, consider we have a square matrix with n rows and n columns. Each cell of the matrix contains a number. Let's denote by c_{ij} the number which lays on the intersection of i-th row and j-th column of the matrix. The task is to choose a subset of the numbers from the matrix in such a way that each row and each column has exactly one number chosen and sum of the chosen numbers is as minimal as possible. For example, assume we had a matrix like this:

$$\begin{pmatrix} 1 & \overset{*}{3} & 2 \\ 2 & 6 & \overset{*}{4} \\ \overset{*}{3} & 7 & 6 \end{pmatrix}$$

In this case, we would chose numbers 3, 4, and 3 with sum 10. In other words, we have to find an integral solution of the following linear programming problem:

$$\text{Minimize } z(x) = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$$

subject to

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \text{for all } i = 1,\ldots,n,$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \text{for all } j = 1,\ldots,n,$$

$$0 \le x_{ij} \le 1,\ x_{ij} \text{ is integral} \quad \text{for all } i,j = 1,\ldots,n.$$

If binary variable $x_{ij} = 1$ we will choose the number from cell (i,j) of the given matrix. Constraints guarantee that each row and each column of the matrix will have only one number chosen. Evidently, the problem has a feasible solution (one can choose all diagonal numbers). To find the optimal solution of the problem we construct the bipartite transportation network as it is drawn in Figure 1. Each edge (i,j') of the graph has unit capacity and cost c_{ij}. All supplies and demands are equal to 1 and -1 respectively. Implicitly, minimum cost flow solution corresponds to the optimal assignment and vise versa. Thanks to *left-to-right* directed edges the network contains no negative cycles and one is able to solve it with complexity of $O(n^3)$. Why? *Hint: use the successive shortest path algorithm.*

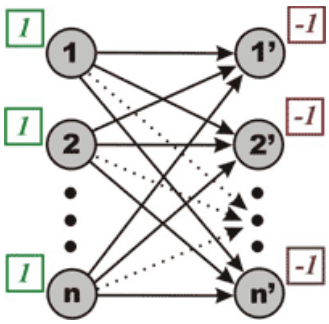


Figure 1. Full weighted bipartite network for the assignment problem. Each edge has capacity 1 and cost according to the number in the given matrix.

The assignment problem can also be represented as weight matching in a weighted bipartite graph. The problem allows some extensions:

- Suppose that there is a different number of supply and demand nodes. The objective might be to find a maximum matching with a minimum weight.
- Suppose that we have to choose not one but k numbers in each row and each column. We could easily solve this task if we considered supplies and demands to be equal to k and -k (instead of 1 and -1) respectively.

However, we should point out that, due to the specialty of the assignment problem, there are more effective algorithms to solve it. For instance, the Hungarian algorithm has complexity of $O(n^3)$, but it works much more quickly in practice.

Discrete Location Problems

Suppose we have n building sites and we have to build n new facilities on these sites. The new facilities interact with m existing facilities. The objective is to assign each new facility i to the available building site j in such a way that minimizes the total transportation cost between the new and existing facilities. One example is the location of hospitals, fire stations etc. in the city; in this case we can treat population concentrations as the existing facilities.

Let's denote by d_{kj} the distance between existing facility k and site j; and the total transportation cost per unit distance between the new facility i and the existing one k by w_{ik}. Let's denote the assignment by binary variable x_{ij}. Given an assignment x we can get a corresponding transportation cost between the new facility i and the existing facility k:

$$w_{ik}\sum_{j=1}^{n} d_{kj}x_{ij}$$

Thus the total transportation cost is given by

$$z(x)=\sum_{i=1}^{n}\sum_{k=1}^{m} w_{ik}\sum_{j=1}^{n} d_{kj}x_{ij} = \sum_{i=1}^{n}\sum_{j=1}^{n}\left(\sum_{k=1}^{m} w_{ik}d_{kj}\right)x_{ij}$$

Note, that $c_{ij} = \sum_{k=1}^{m} w_{ik}d_{kj}$ is the cost of locating the new facility i at site j. Appending necessary conditions, we obtain another instance of the assignment problem.

The Transportation Problem

A minimum cost flow problem is well known to be a *transportation problem in the statement of network*. But there is a special case of transportation problem which is called *the transportation problem in statement of matrix*. We can obtain the optimization model for this case as follows.

$$\text{Minimize } z(x)=\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij}$$

subject to

$$\sum_{j=1}^{n} x_{ij} = b_i \quad \text{for all } i = 1,\ldots,m,$$

$$\sum_{i=1}^{m} x_{ij} = d_j \quad \text{for all } j = 1,\ldots,n,$$

$$0 \le x_{ij} \le u_{ij}, \quad \text{for all } i = 1,\ldots,m \text{ and } j = 1,\ldots,n$$

For example, suppose that we have a set of m warehouses and a set of n shops. Each warehouse i has nonnegative supply value b_i while each shop j has nonnegative demand value d_j. We are able to transfer goods from a warehouse i directly to a shop j by the cost c_{ij} per unit of flow.

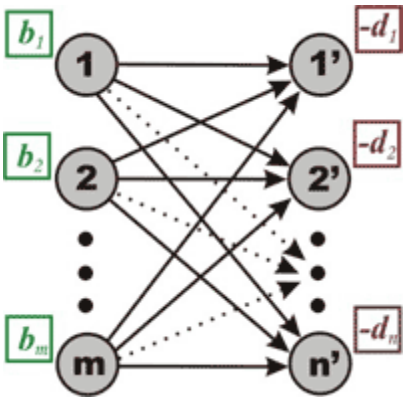


Figure 2. Formulating the transportation problem as a minimum cost flow problem. Each edge connecting a vertex i and a vertex j' has capacity u_{ij} and cost c_{ij}.

There is an upper bound to the amount of flow between each warehouse i and each shop j denoted by u_{ij}. Minimizing total transportation cost is the object. Representing the flow from a warehouse i to a shop j by x_{ij} we obtain the model above. Evidently, the assignment problem is a special case of the transportation problem in the statement of matrix, which in turn is a special case of the minimum cost flow problem.

Optimal Loading of a Hopping Airplane

We took this application from [1]. A small commuter airline uses a plane with the capacity to carry at most p passengers on a "hopping flight." The hopping flight visits the cities 1, 2, …, n, in a fixed sequence. The plane can pick up passengers at any node and drop them off at any other node.

Let b_{ij} denote the number of passengers available at node i who want to go to node j, and let f_{ij} denote the fare per passenger from node i to node j.

The airline would like to determine the number of passengers that the plane should carry between the various origins and destinations in order to maximize the total fare per trip while never exceeding the plane capacity.

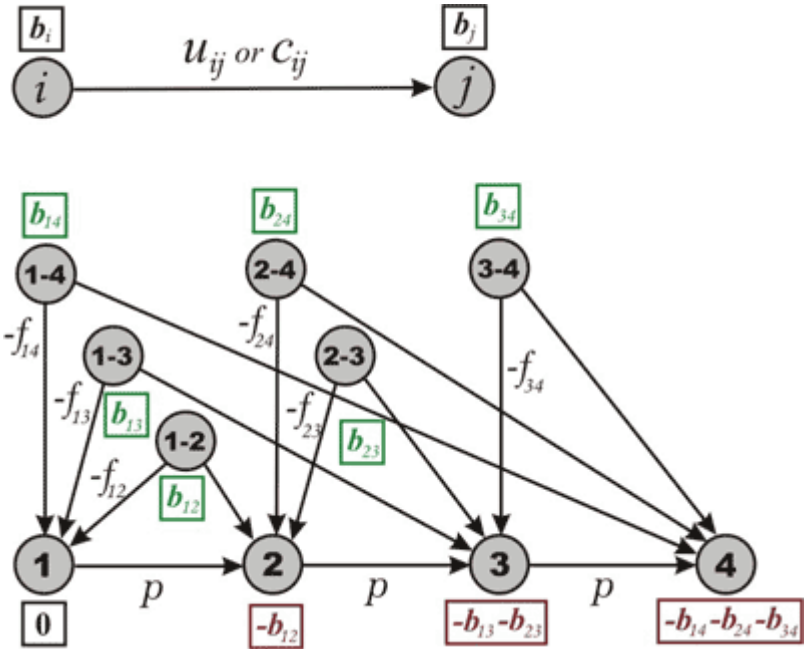


Figure 3. Formulating the hopping plane flight problem as a minimum cost flow problem.

Figure 3 shows a minimum cost flow formulation of this hopping plane flight problem. The network contains data for only those arcs with nonzero costs and with finite capacities: Any arc without an associated cost has a zero cost; any arc without an associated capacity has an infinite capacity.

Consider, for example, node 1. Three types of passengers are available at node 1, those whose destination is node 2, node 3, or node 4. We represent these three types of passengers by the nodes 1-2, 1-3, and 1-4 with supplies b_{12}, b_{13}, and b_{14}. A passenger available at any such node,

say 1-3, either boards the plane at its origin node by flowing though the arc (1-3,1) and thus incurring a cost of -f13 units, or never boards the plane which we represent by the flow through the arc (1-3,3).

We invite the reader to establish one-to-one correspondence between feasible passenger routings and feasible flows in the minimum cost flow formulation of the problem.

Dynamic Lot Sizing

Here's another application that was first outlined in [1]. In the dynamic lot-size problem, we wish to meet prescribed demand d_j for each of K periods j = 1, 2, …, K by either producing an amount a_j in period j and/or by drawing upon the inventory I_{j-1} carried from the previous period. Figure 4 shows the network for modeling this problem.

The network has K+1 vertexes: The j-th vertex, for j = 1, 2, …, K, represents the j-th planning period; node 0 represents the "source" of all production. The flow on the "production arc" (0,j) prescribes the production level a_j in period j, and the flow on "inventory carrying arc" (j,j+1) prescribes the inventory level I_j to be carried from period j to period j+1.

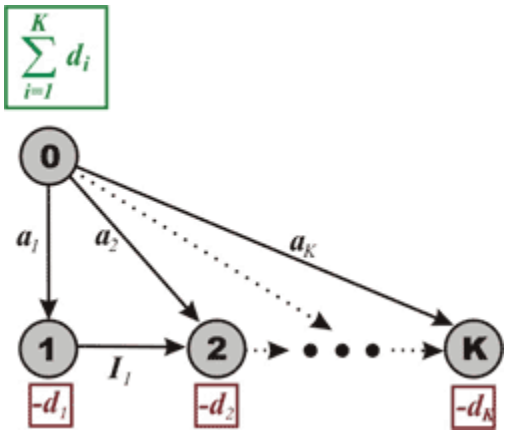


Figure 4. Network flow model of the dynamic lot-size problem.

The mass balance equation for each period j models the basic accounting equation: Incoming inventory plus production in that period must equal the period's demand plus the final inventory at the end of the period. The mass balance equation for vertex 0 indicates that during the planning periods 1, 2, …, K, we must produce all of the demand (we are assuming zero beginning and zero final inventory over the planning horizon).

If we impose capacities on the production and inventory in each period and suppose that the costs are linear, the problem becomes a minimum cost flow model.

Introduction

The games we will talk about are two-person games with perfect information, no chance moves, and a win-or-lose outcome. In these games, players usually alternate moves until they reach a terminal position. After that, one player is declared the winner and the other the loser. Most card games don't fit this category, for example, because we do not have information about what cards our opponent has.

First we will look at the basic division of positions to winning and losing. After that we will master the most important game -- the Game of Nim -- and see how understanding it will help us to play composite games. We will not be able to play many of the games without decomposing them to smaller parts (sub-games), pre-computing some values for them, and then obtaining the result by combining these values.

The Basics

A simple example is the following game, played by two players who take turns moving. At the beginning there are n coins. When it is a player's turn he can take away 1, 3 or 4 coins. The player who takes the last one away is declared the winner (in other words, the player who can not make a move is the loser). The question is: For what n will the first player win if they both play optimally?

We can see that n = 1, 3, 4 are winning positions for the first player, because he can simply take all the coins. For n=0 there are no possible moves -- the game is finished -- so it is the losing position for the first player, because he can not make a move from it. If n=2 the first player has only one option, to remove 1 coin. If n=5 or 6 a player can move to 2 (by removing 3 or 4 coins), and he is in a winning position. If n=7 a player can move only to 3, 4, 6, but from all of them his opponent can win…

Positions have the following properties:

- All terminal positions are losing.
- If a player is able to move to a losing position then he is in a winning position.
- If a player is able to move only to the winning positions then he is in a losing position.

These properties could be used to create a simple recursive algorithm **WL-Algorithm**:

```
boolean isWinning(position pos) {
    moves[] = possible positions to which I can move from the
position pos;
    for (all x in moves)
        if (!isWinning(x)) return true;

    return false;
}
```

Table 1: Game with 11 coins and subtraction set {1, 3, 4}:

n	0	1	2	3	4	5	6	7	8	9	10	11
position	L	W	L	W	W	W	W	L	W	L	W	W

This game could be played also with a rule (usually called the misere play rule) that the player who takes away the last coin is declared the loser. You need to change only the behavior for the terminal positions in WL-Algorithm. Table 1 will change to this:

n	0	1	2	3	4	5	6	7	8	9	10	11
position	W	L	W	L	W	W	W	W	L	W	L	W

It can be seen that whether a position is winning or losing depends only on the last k positions, where k is the maximum number of coins we can take away. While there are only 2^k possible values for the sequences of the length k, our sequence will become periodic. You can try to use this observation to solve the following problem:
SRM 330: LongLongNim

The Game of Nim

The most famous mathematical game is probably the Game of Nim. This is the game that you will probably encounter the most times and there are many variations on it, as well as games that can be solved by using the knowledge of how to play the game. Usually you will meet them as

Division I 1000 pointers (though hopefully your next encounter will seem much easier). Although these problems often require a clever idea, they are usually very easy to code.

Rules of the Game of Nim: There are n piles of coins. When it is a player's turn he chooses one pile and takes at least one coin from it. If someone is unable to move he loses (so the one who removes the last coin is the winner).

Let n1, n2, … nk, be the sizes of the piles. It is a losing position for the player whose turn it is if and only if $\mathbf{n_1}$ **xor** $\mathbf{n_2}$ **xor .. xor** $\mathbf{n_k = 0}$.

How is xor being computed?

```
6 = (110)2        1 1 0
9 = (1001)2       1 0 0 1
3 = (11)2          1 1
          --------
          1 1 0 0
```

- **xor** of two logic values is true if and only if one of them is true and the second is false
- when computing **xor** of integers, first write them as binary numbers and then apply **xor** on columns.
- so **xor** of even number of 1s is 0 and **xor** of odd number of 1s is 1

Why does it work?

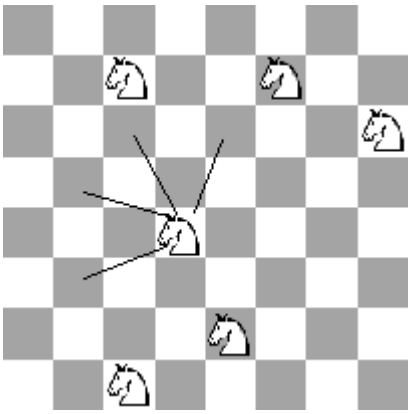
- From the losing positions we can move only to the winning ones:
  - if xor of the sizes of the piles is 0 then it will be changed after our move (at least one 1 will be changed to 0, so in that column will be odd number of 1s).
- From the winning positions it is possible to move to at least one losing:
  - if xor of the sizes of the piles is not 0 we can change it to 0 by finding the left most column where the number of 1s is odd, changing one of them to 0 and then by changing 0s or 1s on the right side of it to gain even number of 1s in every column.

Examples:
Position (1, 2, 3) is losing because 1 xor 2 xor 3 = $(1)_2$ xor $(10)_2$ xor $(11)_2$ = 0
Position (7, 4, 1) is winning because 7 xor 4 xor 1 = $(111)_2$ xor $(10)_2$ xor $(1)_2$ = $(10)_2$ = 2

Example problems:
SRM 338: CakeParty
SRM 309: StoneGameStrategist

The last one example problem is harder, because it is not so easy to identify where the sizes of piles are hidden. Small hint: Notice the differences between the sizes of piles. If you would not be able to figure it out you can find the solution in the SRM 309 Problem set & Analysis.

Composite games - Grundy numbers
Example game: N x N chessboard with K knights on it. Unlike a knight in a traditional game of chess, these can move only as shown in the picture below (so the sum of coordinates is decreased in every move). There can be more than one knight on the same square at the same time. Two players take turns moving and, when it is a player's, turn he chooses one of the knights and moves it. A player who is not able to make a move is declared the loser.



This is the same as if we had K chessboards with exactly one knight on every chessboard. This is the ordinary sum of K games and it can be solved by using the grundy numbers. We assign grundy number to every subgame according to which size of the pile in the Game of Nim it is equivalent to. When we know how to play Nim we will be able to play this game as well.

```
int grundyNumber(position pos) {
    moves[] = possible positions to which I can move from pos
    set s;
    for (all x in moves) insert into s grundyNumber(x);
    //return the smallest non-negative integer not in the set s;
    int ret=0;
    while (s.contains(ret)) ret++;
    return ret;
}
```

The following table shows grundy numbers for an 8 x 8 board:

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

We could try to solve the original problem with our WL-Algorithm, but it would time out because of the large number of possible positions.

A better approach is to compute grundy numbers for an N x N chessboard in $O(n^2)$ time and then xor these K (one for every horse) values. If their xor is 0 then we are in a losing position, otherwise we are in a winning position.

Why is the pile of Nim equivalent to the subgame if its size is equal to the grundy number of that subgame?

- If we decrease the size of the pile in Nim from A to B, we can move also in the subgame to the position with the grundy number B. (Our current position had grundy number A so it means we could move to positions with all smaller grundy numbers, otherwise the grundy number of our position would not be A.)
- If we are in the subgame at a position with a grundy number higher than 0, by moving in it and decreasing its grundy number we can also decrease the size of pile in the Nim.
- If we are in the subgame at the position with grundy number 0, by moving from that we will get to a position with a grundy number higher than 0. Because of that, from such a position it is possible to move back to 0. By doing that we can nullify every move from the position from grundy number 0.

Example problems:
SRM 216: Roxor

Other composite games
It doesn't happen often, but you can occasionally encounter games with a slightly different set of rules. For example, you might see the following changes:

1. When it is a player's move he can choose some of the horses (at least one) and move with all the chosen ones.

Solution: You are in a losing position if and only if every horse is in a losing position on his own chessboard (so the grundy number for every square, where the horse is, is 0).

2. When it is a player's move he can choose some of the horses (at least one), but not all of them, and move with all chosen ones.

Solution: You are in a losing position if and only if the grundy numbers of all the positions, where horses are, are the same.

You can verify correctness of both solutions by verifying the basic properties (from a winning position it is possible to move to a losing one and from a losing position it is possible to move only to the winning ones). Of course, everything works for all other composite games with these rules (not only for horse games).

Homework: What would be changed if a player had to move with every horse and would lose if he were not able to do so?

Introduction

We often need some sort of data structure to make our algorithms faster. In this article we will discuss the **Binary Indexed Trees** structure. According to Peter M. Fenwick, this structure was first used for data compression. Now it is often used for storing frequencies and manipulating cumulative frequency tables.

Let's define the following **problem**: We have n boxes. Possible queries are
1. add marble to box **i**
2. sum marbles from box **k** to box **l**

The naive solution has time complexity of O(1) for query 1 and O(n) for query 2. Suppose we make **m** queries. The worst case (when all queries are 2) has time complexity O(n * m). Using some data structure (i.e. RMQ) we can solve this problem with the worst case time complexity of O(m log n). Another approach is to use Binary Indexed Tree data structure, also with the worst time complexity O(m log n) -- but Binary Indexed Trees are much easier to code, and require less memory space, than RMQ.

Notation

BIT - **B**inary **I**ndexed **T**ree
MaxVal - maximum value which will have non-zero frequency
f[i] - frequency of value with index **i**, **i** = 1 .. MaxVal
c[i] - cumulative frequency for index **i** (f[1] + f[2] + ... + f[i])
tree[i] - sum of frequencies stored in **BIT** with index **i** (latter will be described what index means); sometimes we will write *tree frequency* instead *sum of frequencies stored in BIT*
num^{-} - complement of integer **num** (integer where each binary digit is inverted: 0 -> 1; 1 -> 0)
NOTE: Often we put f[0] = 0, c[0] = 0, tree[0] = 0, so sometimes I will just ignore index 0.

Basic idea

Each integer can be represented as sum of powers of two. In the same way, cumulative frequency can be represented as sum of sets of subfrequencies. In our case, each set contains some successive number of non-overlapping frequencies.

idx is some index of **BIT**. **r** is a position in **idx** of the last digit 1 (from left to right) in binary notation. **tree[idx]** is sum of frequencies from index (**idx** - 2^**r** + 1) to index **idx** (look at the Table 1.1 for clarification). We also write that **idx** is **responsible** for indexes from (**idx** - 2^**r** + 1) to **idx** (note that responsibility is the key in our algorithm and is the way of manipulating the tree).

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
f	1	0	2	1	1	3	0	4	2	5	2	2	3	1	0	2
c	1	1	3	4	5	8	8	12	14	19	21	23	26	27	27	29
tree	1	1	2	4	1	4	0	12	2	7	2	11	3	4	0	29

Table 1.1

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
tree	1	1..2	3	1..4	5	5..6	7	1..8	9	9..10	11	9..12	13	13..14	15	1..16

Table 1.2 - table of responsibility

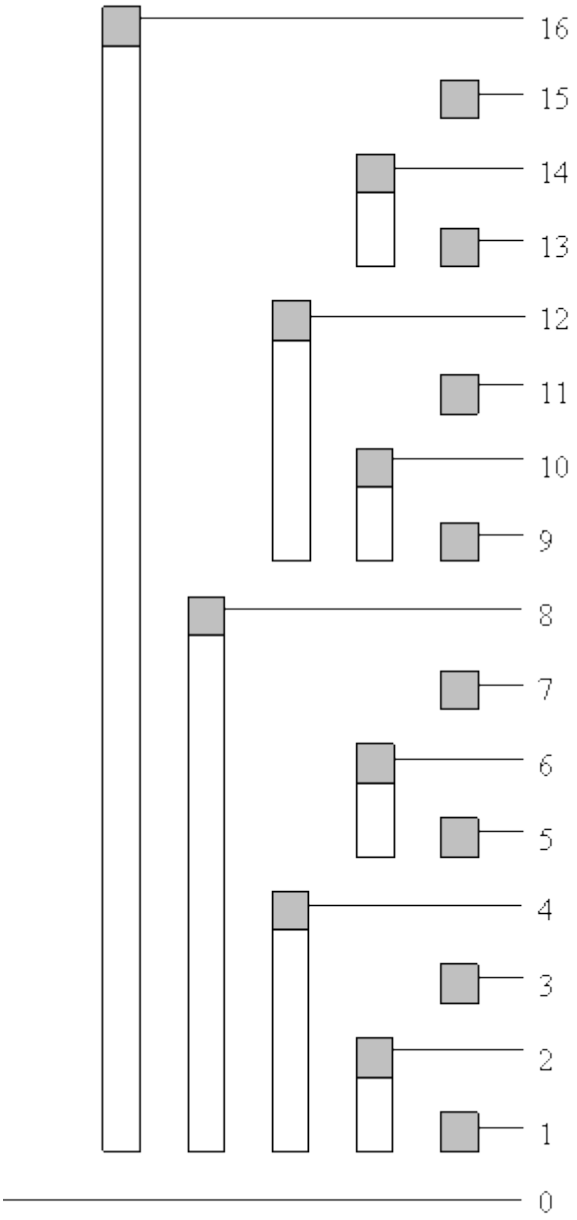


Image 1.3 - tree of responsibility for indexes (bar shows range of frequencies accumulated in top element)

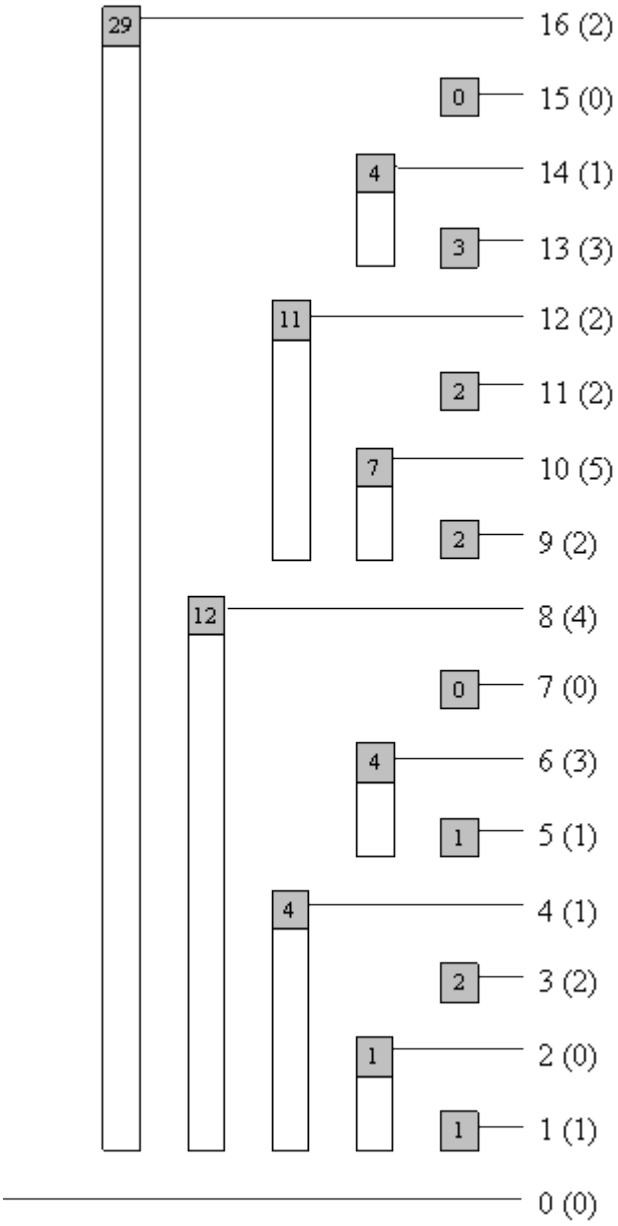


Image 1.4 - tree with tree frequencies

Suppose we are looking for cumulative frequency of index 13 (for the first 13 elements). In binary notation, 13 is equal to 1101. Accordingly, we will calculate **c[1101] = tree[1101] + tree[1100] + tree[1000]** (more about this later).

Isolating the last digit
NOTE: Instead of "the last non-zero digit," it will write only "the last digit."

There are times when we need to get just the last digit from a binary number, so we need an efficient way to do that. Let **num** be the integer whose last digit we want to isolate. In binary notation **num** can be represented as **a1b**, where **a** represents binary digits before the last digit and **b** represents zeroes after the last digit.

Integer **-num** is equal to $(a1b)^- + 1 = a^-0b^- + 1$. **b** consists of all zeroes, so b^- consists of all ones. Finally we have

$$-num = (a1b)^- + 1 = a^-0b^- + 1 = a^-0(0...0)^- + 1 = a^-0(1...1) + 1 = a^-1(0...0) = a^-1b.$$

Now, we can easily isolate the last digit, using bitwise operator **AND** (in C++, Java it is **&**) with **num** and **-num**:

a1b
& a¯1b

= (0...0)1(0...0)

Read cumulative frequency
If we want to read cumulative frequency for some integer **idx**, we add to **sum tree[idx]**, substract last bit of **idx** from itself (also we can write - remove the last digit; change the last digit to zero) and repeat this while **idx** is greater than zero. We can use next function (written in C++)

```
int read(int idx){
        int sum = 0;
        while (idx > 0){
                sum += tree[idx];
                idx -= (idx & -idx);
        }
        return sum;
}
```

Example for **idx** = 13; **sum** = 0:

iteration	idx	position of the last digit	idx & -idx	sum
1	13 = 1101	0	1 (2 ^0)	3
2	12 = 1100	2	4 (2 ^2)	14
3	8 = 1000	3	8 (2 ^3)	26
4	0 = 0	---	---	---

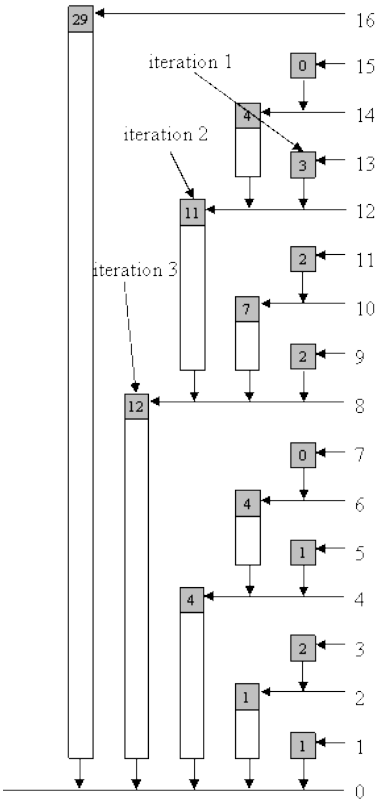


Image 1.5 - arrows show path from index to zero which we use to get sum (image shows example for index 13)

So, our result is 26. The number of iterations in this function is number if bits in **idx**, which is at most **log MaxVal**.

Time complexity: O(log MaxVal).
Code length: Up to ten lines.

Change frequency at some position and update tree
The concept is to update tree frequency at all indexes which are responsible for frequency whose value we are changing. In reading cumulative frequency at some index, we were removing the last bit and going on. In changing some frequency **val** in tree, we should increment value at the current index (the starting index is always the one whose frequency is changed) for **val**, add the last digit to index and go on while the index is less than or equal to **MaxVal**. Function in C++:

```
void update(int idx ,int val){
        while (idx <= MaxVal){
                tree[idx] += val;
                idx += (idx & -idx);
        }
}
```

Let's show example for **idx** = 5:

iteration	idx	position of the last digit	idx & -idx
1	5 = 101	0	1 (2 ^0)
2	6 = 110	1	2 (2 ^1)
3	8 = 1000	3	8 (2 ^3)
4	16 = 10000	4	16 (2 ^4)
5	32 = 100000	---	---

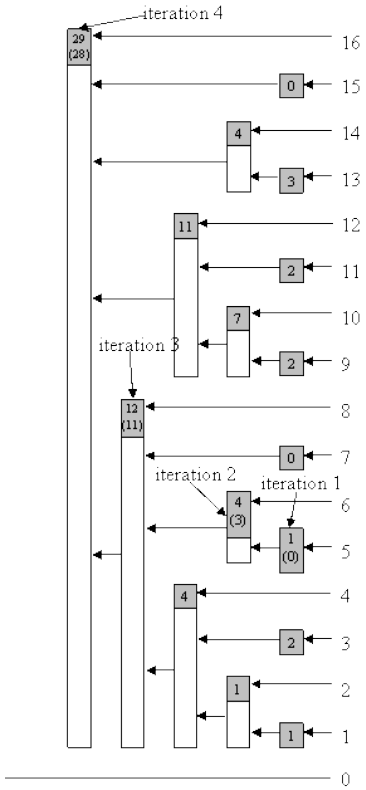


*Image 1.6 - Updating tree (in brackets are tree frequencies before updating); arrows show path while we update tree from index to **MaxVal** (image shows example for index 5)*

Using algorithm from above or following arrows shown in Image 1.6 we can update **BIT**.

Time complexity: O(log MaxVal).
Code length: Up to ten lines.

Read the actual frequency at a position

We've described how we can read cumulative frequency for an index. It is obvious that we can not read just **tree[idx]** to get the actual frequency for value at index **idx**. One approach is to have one aditional array, in which we will seperately store frequencies for values. Both reading and storing take O(1); memory space is linear. Sometimes it is more important to save memory, so we will show how you can get actual frequency for some value without using aditional structures.

Probably everyone can see that the actual frequency at a position **idx** can be calculated by calling function **read** twice -- **f[idx] = read(idx) - read(idx - 1)** -- just by taking the difference of two adjacent cumulative frequencies. This procedure always works in 2 * O(log n) time. If we write a new function, we can get a bit faster algorithm, with smaller const.

If two paths from two indexes to root have the same part of path, then we can calculate the sum until the paths meet, substract stored sums and we get a sum of frequencies between that two indexes. It is pretty simple to calculate sum of frequencies between adjacent indexes, or read the actual frequency at a given index.

Mark given index with **x**, its predecessor with **y**. We can represent (binary notation) **y** as **a0b**, where **b** consists of all ones. Then, **x** will be **a1b**$^{-}$ (note that **b**$^{-}$ consists all zeros). Using our algorithm for getting **sum** of some index, let it be **x**, in first iteration we remove the last digit, so after the first iteration **x** will be **a0b**$^{-}$, mark a new value with **z**.

Repeat the same process with **y**. Using our function for reading **sum** we will remove the last digits from the number (one by one). After several steps, our **y** will become (just to remind, it was **a0b**) **a0b**$^{-}$, which is the same as **z**. Now, we can write our algorithm. Note that the only exception is when **x** is equal to 0. Function in C++:

```
int readSingle(int idx){
int sum = tree[idx]; // sum will be decreased
if (idx > 0){ // special case
        int z = idx - (idx & -idx); // make z first
        idx--; // idx is no important any more, so instead y, you can use idx
        while (idx != z){ // at some iteration idx (y) will become z
                sum -= tree[idx];
// substruct tree frequency which is between y and "the same path"
                idx -= (idx & -idx);
        }
}
return sum;
}
```

Here's an example for getting the actual frequency for index 12:

First, we will calculate **z = 12 - (12 & -12) = 8**, **sum = 11**

iteration	y	position of the last digit	y & -y	sum
1	11 = 1011	0	1 (2 ^0)	9
2	10 = 1010	1	2 (2 ^1)	2
3	8 = 1000	---	---	---

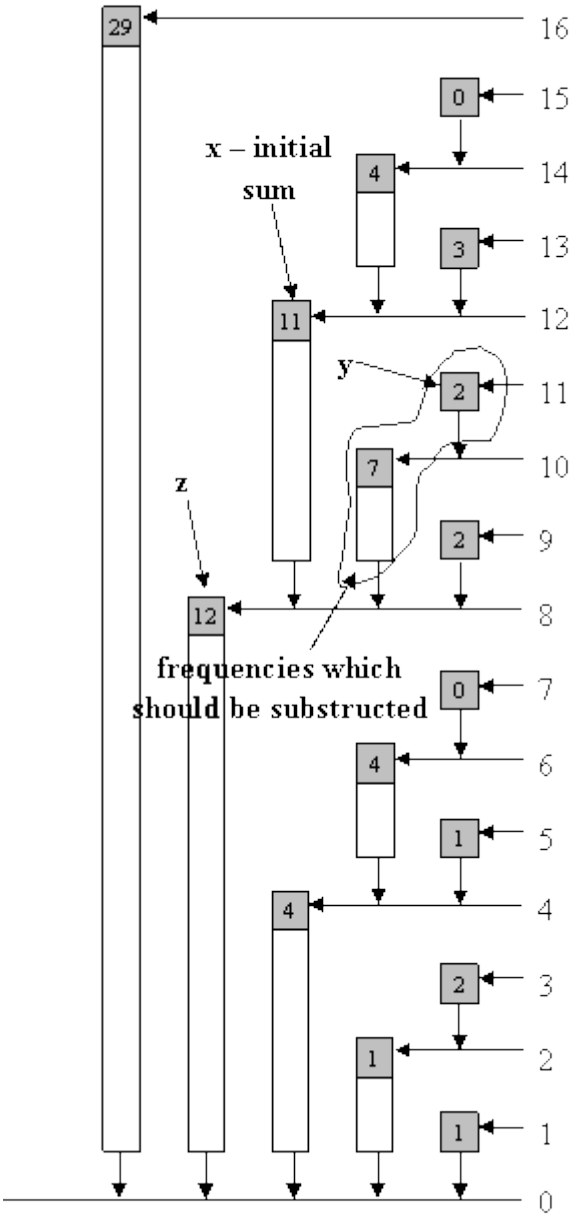


Image 1.7 - read actual frequency at some index in BIT
(image shows example for index 12)

Let's compare algorithm for reading actual frequency at some index when we twice use function **read** and the algorithm written above. Note that for each odd number, the algorithm will work in const time O(1), without any iteration. For almost every even number **idx**, it will work in c * O(log idx), where c is strictly less than 1, compare to **read(idx) - read(idx - 1),** which will work in c1 * O(log idx), where c1 is **always** greater than 1.

Time complexity: c * O(log MaxVal), where c is less than 1.
Code length: Up to fifteen lines.

Scaling the entire tree by a constant factor
Sometimes we want to scale our tree by some factor. With the procedures described above it is very simple. If we want to scale by some factor **c**, then each index **idx** should be updated by **-(c - 1) * readSingle(idx) / c** (because **f[idx] - (c - 1) * f[idx] / c = f[idx] / c**). Simple function in C++:

```
void scale(int c){
        for (int i = 1 ; i <= MaxVal ; i++)
                update(-(c - 1) * readSingle(i) / c , i);
}
```

This can also be done more quickly. Factor is linear operation. Each tree frequency is a linear composition of some frequencies. If we scale each frequency for some factor, we also scaled tree frequency for the same factor. Instead of rewriting the procedure above, which has time complexity O(MaxVal * log MaxVal), we can achieve time complexity of O(MaxVal):

```
void scale(int c){
        for (int i = 1 ; i <= MaxVal ; i++)
                tree[i] = tree[i] / c;
}
```

Time complexity: O(MaxVal).
Code length: Just a few lines.

Find index with given cumulative frequency

The naive and most simple solution for finding an index with a given cumultive frequency is just simply iterating through all indexes, calculating cumulative frequency, and checking if it's equal to the given value. In case of negative frequencies it is the only solution. However, if we have only non-negative frequencies in our tree (that means cumulative frequencies for greater indexes are not smaller) we can figure out logarithmic algorithm, which is modification of binary search. We go through all bits (starting with the highest one), make the index, compare the cumulative frequency of the current index and given value and, according to the outcome, take the lower or higher half of the interval (just like in binary search). Function in C++:

```
// if in tree exists more than one index with a same
// cumulative frequency, this procedure will return
// some of them (we do not know which one)

// bitMask - initialy, it is the greatest bit of MaxVal
// bitMask store interval which should be searched
int find(int cumFre){
        int idx = 0; // this var is result of function

        while ((bitMask != 0) && (idx < MaxVal)){ // nobody likes overflow :)
                int tIdx = idx + bitMask; // we make midpoint of interval
                if (cumFre == tree[tIdx]) // if it is equal, we just return idx
                        return tIdx;
                else if (cumFre > tree[tIdx]){
                        // if tree frequency "can fit" into cumFre,
                        // then include it
                        idx = tIdx; // update index
                        cumFre -= tree[tIdx]; // set frequency for next loop
                }
                bitMask >>= 1; // half current interval
        }
        if (cumFre != 0) // maybe given cumulative frequency doesn't exist
                return -1;
        else
                return idx;
}



// if in tree exists more than one index with a same
// cumulative frequency, this procedure will return
// the greatest one
int findG(int cumFre){
        int idx = 0;

        while ((bitMask != 0) && (idx < MaxVal)){
                int tIdx = idx + bitMask;
                if (cumFre >= tree[tIdx]){
                        // if current cumulative frequency is equal to cumFre,
                        // we are still looking for higher index (if exists)
                        idx = tIdx;
```

```
                        cumFre -= tree[tIdx];
                }
                bitMask >>= 1;
        }
        if (cumFre != 0)
                return -1;
        else
                return idx;
}
```

Example for cumulative frequency 21 and function **find**:

First iteration	tIdx is 16; tree[16] is greater than 21; half bitMask and continue
Second iteration	tIdx is 8; tree[8] is less than 21, so we should include first 8 indexes in result, remember idx because we surely know it is part of result; subtract tree[8] of cumFre (we do not want to look for the same cumulative frequency again - we are looking for another cumulative frequency in the rest/another part of tree); half bitMask and contiue
Third iteration	tIdx is 12; tree[12] is greater than 9 (there is no way to overlap interval 1-8, in this example, with some further intervals, because only interval 1-16 can overlap); half bitMask and continue
Forth iteration	tIdx is 10; tree[10] is less than 9, so we should update values; half bitMask and continue
Fifth iteration	tIdx is 11; tree[11] is equal to 2; return index (tIdx)

Time complexity: O(log MaxVal).
Code length: Up to twenty lines.

2D BIT
BIT can be used as a multi-dimensional data structure. Suppose you have a plane with dots (with non-negative coordinates). You make three queries:

1. set dot at (x , y)
2. remove dot from (x , y)
3. count number of dots in rectangle (0 , 0), (x , y) - where (0 , 0) if down-left corner, (x , y) is up-right corner and sides are parallel to x-axis and y-axis.

If **m** is the number of queries, **max_x** is maximum x coordinate, and **max_y** is maximum y coordinate, then the problem should be solved in O(m * log (max_x) * log (max_y)). In this case, each element of the tree will contain array - (**tree[max_x][max_y]**). Updating indexes of x-coordinate is the same as before. For example, suppose we are setting/removing dot (**a** , **b**). We will call **update(a , b , 1)/update(a , b , -1)**, where **update** is:

```
void update(int x , int y , int val){
        while (x <= max_x){
                updatey(x , y , val);
                // this function should update array tree[x]
                x += (x & -x);
        }
}
```

The function **updatey** is the "same" as function **update**:

```
void updatey(int x , int y , int val){
        while (y <= max_y){
                tree[x][y] += val;
                y += (y & -y);
        }
}
```

It can be written in one function/procedure:

```
void update(int x , int y , int val){
        int y1;
        while (x <= max_x){
                y1 = y;
                while (y1 <= max_y){
                        tree[x][y1] += val;
                        y1 += (y1 & -y1);
                }
                x += (x & -x);
        }
}
```

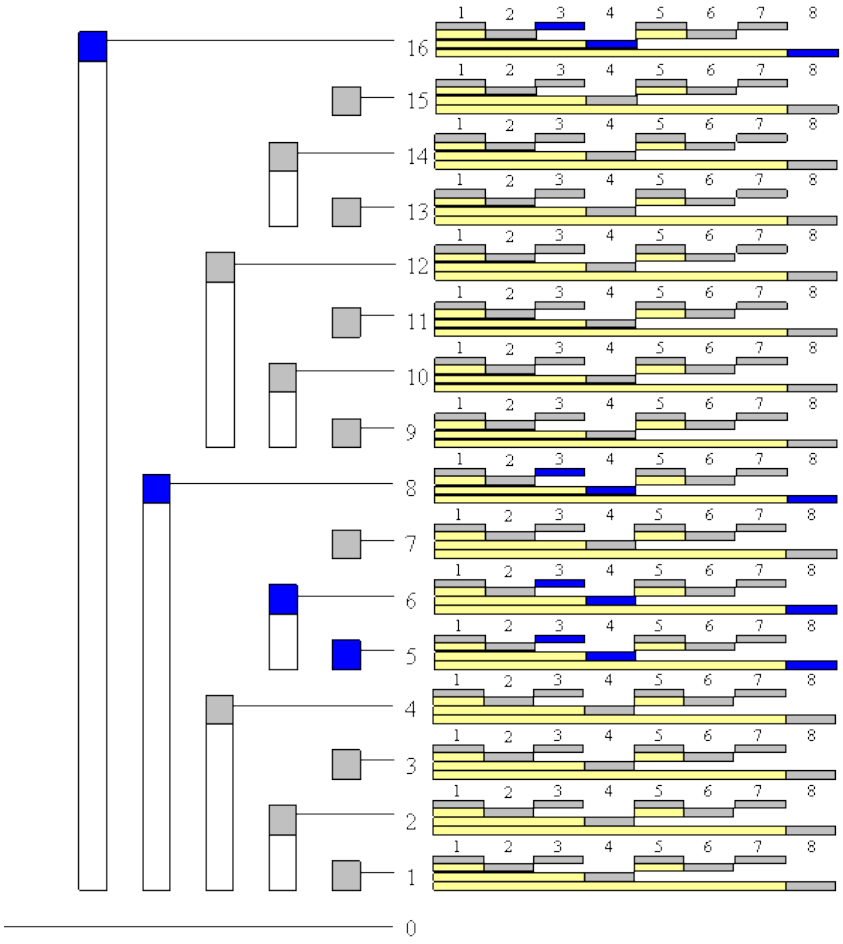


Image 1.8 - BIT is array of arrays, so this is two-dimensional BIT (size 16 x 8).
Blue fields are fields which we should update when we are updating index (5 , 3).

The modification for other functions is very similar. Also, note that BIT can be used as an n-dimensional data structure.

Sample problem

- SRM 310 - FloatingMedian
- Problem 2:
  Statement:
  There is an array of **n** cards. Each card is putted face down on table. You have two queries:
  1. T i j (turn cards from index i to index j, include i-th and j-th card - card which was face down will be face up; card which was face up

will be face down)
2. Q i (answer 0 if i-th card is face down else answer 1)

Solution:
This has solution for each query (and 1 and 2) has time complexity O(log n). In array **f** (of length **n + 1**) we will store each query **T (i , j)** - we set **f[i]++** and **f[j + 1]--**. For each card **k** between **i** and **j** (include **i** and **j**) sum **f[1] + f[2] + ... + f[k]** will be increased for 1, for all others will be same as before (look at the image 2.0 for clarification), so our solution will be described sum (which is same as cumulative frequency) module 2.

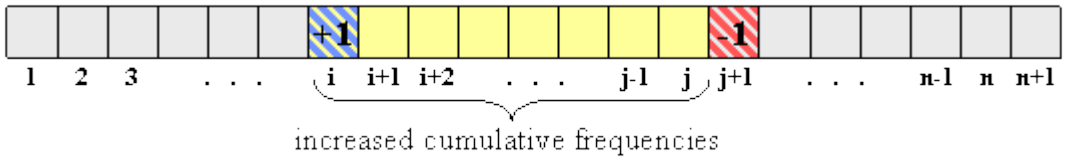


Image 2.0

Use **BIT** to store (increase/decrease) frequency and read cumulative frequency.

Conclusion

- Binary Indexed Trees are very easy to code.
- Each query on Binary Indexed Tree takes constant or logarithmic time.
- Binary Indexeds Tree require linear memory space.
- You can use it as an n-dimensional data structure.

The fundamental string searching (matching) problem is defined as follows: given two strings - a text and a pattern, determine whether the pattern appears in the text. The problem is also known as "the needle in a haystack problem."

The "Naive" Method
Its idea is straightforward -- for every position in the text, consider it a starting position of the pattern and see if you get a match.

```
function brute_force(text[], pattern[])
{
  // let n be the size of the text and m the size of the
  // pattern

  for(i = 0; i < n; i++) {
    for(j = 0; j < m && i + j < n; j++)
      if(text[i + j] != pattern[j]) break;
      // mismatch found, break the inner loop
    if(j == m) // match found
  }
}
```

The "naive" approach is easy to understand and implement but it can be too slow in some cases. If the length of the text is n and the length of the pattern m, in the worst case it may take as much as (n * m) iterations to complete the task.

It should be noted though, that for most practical purposes, which deal with texts based on human languages, this approach is much faster since the inner loop usually quickly finds a mismatch and breaks. A problem arises when we are faced with different kinds of "texts," such as the genetic code.

Rabin-Karp Algorithm (RK)
This is actually the "naive" approach augmented with a powerful programming technique - the hash function.

Every string s[] of length m can be seen as a number H written in a positional numeral system in base B (B >= size of the alphabet used in the string):

$$H = s[0] * B^{(m-1)} + s[1] * B^{(m-2)} + \ldots + s[m-2] * B^{1} + s[m-1] * B^{0}$$

If we calculate the number H (the hash value) for the pattern and the same number for every substring of length m of the text than the inner loop of the "naive" method will disappear - instead of comparing two strings character by character we will have just to compare two integers.

A problem arises when m and B are big enough and the number H becomes too large to fit into the standard integer types. To overcome this, instead of the number H itself we use its remainder when divided by some other number M. To get the remainder we do not have to calculate H. Applying the basic rules of modular arithmetic to the above expression:

$$A + B = C \Rightarrow (A \% M + B \% M) \% M = C \% M$$
$$A * B = C \Rightarrow ((A \% M) * (B \% M)) \% M = C \% M$$

We get:

$$H \% M = (((s[0] \% M) * (B^{(m-1)} \% M)) \% M + ((s[1] \% M) * (B^{(m-2)} \% M)) \% M + \ldots$$
$$\ldots + ((s[m-2] \% M) * (B^{1} \% M)) \% M + ((s[m-1] \% M) * (B^{0} \% M)) \% M) \% M$$

The drawback of using remainders is that it may turn out that two different strings map to the same number (it is called a collision). This is less likely to happen if M is sufficiently large and B and M are prime numbers. Still this does not allow us to entirely skip the inner loop of the "naive" method. However, its usage is significantly limited. We have to compare the "candidate" substring of the text with the pattern character by character only when their hash values are equal.

Obviously the approach described so far would be absolutely useless if we were not able to calculate the hash value for every substring of length m in the text in just one pass through the entire text. At first glance to do these calculations we will again need two nested loops: an outer one -- to iterate through all possible starting positions -- and an inner one -- to calculate the hash function for every starting position. Fortunately, this is not the case. Let's consider a string s[], and let's suppose we are to calculate the hash value for every substring in s[] with length say m = 3. It is easy to see that:

$$H_0 = H_{s[0]\ldots s[2]} = s[0] * B^{2} + s[1] * B + s[2]$$

$H_1 = H_{s[1]..s[3]} = s[1] * B^2 + s[2] * B + s[3]$

$H_1 = (H_0 - s[0] * B^2) * B + s[3]$

In general:

$H_i = (H_{i-1} - s[i- 1] * B^{m-1}) * B + s[i + m - 1]$

Applying again the rules of modular arithmetic, we get:

$H_i \% M = ((((H_{i-1} \% M - ((s[i- 1] \% M) * (B^{m-1} \% M)) \% M) \% M) * (B \% M)) \% M +$
$+ s[i + m - 1] \% M) \% M$

Obviously the value of (H_{i-1} - s[i - 1] * B^{m-1}) may be negative. Again, the rules of modular arithmetic come into play:

A - B = C => (A % M - B % M + k * M) % M = C % M

Since the absolute value of (H_{i-1} - s[i - 1] * B^{m-1}) is between 0 and (M - 1), we can safely use a value of 1 for k.

Pseudocode for RK follows:

```
// correctly calculates a mod b even if a < 0
function int_mod(int a, int b)
{
  return (a % b + b) % b;
}

function Rabin_Karp(text[], pattern[])
{
  // let n be the size of the text, m the size of the
  // pattern, B - the base of the numeral system,
  // and M - a big enough prime number

  if(n < m) return; // no match is possible

  // calculate the hash value of the pattern
  hp = 0;
  for(i = 0; i < m; i++)
    hp = int_mod(hp * B + pattern[i], M);

  // calculate the hash value of the first segment
  // of the text of length m
  ht = 0;
  for(i = 0; i < m; i++)
    ht = int_mod(ht * B + text[i], M);

  if(ht == hp) check character by character if the first
               segment of the text matches the pattern;

  // start the "rolling hash" - for every next character in
  // the text calculate the hash value of the new segment
  // of length m; E = (Bm-1) modulo M
  for(i = m; i < n; i++) {
    ht = int_mod(ht - int_mod(text[i - m] * E, M), M);
    ht = int_mod(ht * B, M);
    ht = int_mod(ht + text[i], M);

    if(ht == hp) check character by character if the
                 current segment of the text matches
                 the pattern;
  }
}
```

Unfortunately, there are still cases when we will have to run the entire inner loop of the "naive" method for every starting position in the text -- for example, when searching for the pattern "aaa" in the string "aaaaaaaaaaaaaaaaaaaaaaaa" -- so in the worst case we will still need (n * m) iterations. How do we overcome this?

Let's go back to the basic idea of the method -- to replace the string comparison character by character by a comparison of two integers. In order to keep those integers small enough we have to use modular arithmetic. This causes a "side effect" -- the mapping between strings and integers ceases to be unique. So now whenever the two integers are equal we still have to "confirm" that the two strings are identical by running character-by-character comparison. It can become a kind of vicious circle…

The way to solve this problem is "rational gambling," or the so called "double hash" technique. We "gamble" -- whenever the hash values of two strings are equal, we assume that the strings are identical, and do not compare them character by character. To make the likelihood of a "mistake" negligibly small we compute for every string not one but two independent hash values based on different numbers B and M. If both are equal, we assume that the strings are identical. Sometimes even a "triple hash" is used, but this is rarely justifiable from a practical point of view.

The "pure" form of "the needle in a haystack problem" is considered too straightforward and is rarely seen in programming contests. However, the "rolling hash" technique used in RK is an important weapon. It is especially useful in problems where we have to look at all substrings of fixed length of a given text. An example is "the longest common substring problem": given two strings find the longest string that is a substring of both. In this case, the combination of binary search (BS) and "rolling hash" works quite well. The important point that allows us to use BS is the fact that if the given strings have a common substring of length n, they also have at least one common substring of any length m < n. And if the two strings do not have a common substring of length n they do not have a common substring of any length m > n. So all we need is to run a BS on the length of the string we are looking for. For every substring of the first string of the length fixed in the BS we insert it in a hash table using one hash value as an index and a second hash value ("double hash") is inserted in the table. For every substring of the fixed length of the second string, we calculate the corresponding two hash values and check in the table to see if they have been already seen in the first string. A hash table based on open addressing is very suitable for this task.

Of course in "real life" (real contests) the number of the given strings may be greater than two, and the longest substring we are looking for should not necessarily be present in all the given strings. This does not change the general approach.

Another type of problems where the "rolling hash" technique is the key to the solution are those that ask us to find the most frequent substring of a fixed length in a given text. Since the length is already fixed we do not need any BS. We just use a hash table and keep track of the frequencies.

Knuth-Morris-Pratt Algorithm (KMP)
In some sense, the "naive" method and its extension RK reflect the standard approach of human logic to "the needle in a haystack problem". The basic idea behind KMP is a bit different. Let's suppose that we are able, after one pass through the text, to identify all positions where an existing match with the pattern ends. Obviously, this will solve our problem. Since we know the length of the pattern, we can easily identify the starting position of every match.

Is this approach feasible? It turns out that it is, when we apply the concept of the automaton. We can think of an automaton as of a kind of abstract object, which can be in a finite number of states. At each step some information is presented to it. Depending on this information and its current state the automaton goes to a new state, uniquely determined by a set of internal rules. One of the states is considered as "final". Every time we reach this "final" state we have found an end position of a match.

The automaton used in KMP is just an array of "pointers" (which represents the "internal rules") and a separate "external" pointer to some index of that array (which represents the "current state"). When the next character from the text is presented to the automaton, the position of the "external" pointer changes according to the incoming character, the current position, and the set of "rules" contained in the array. Eventually a "final" state is reached and we can declare that we have found a match.

The general idea behind the automaton is relatively simple. Let us consider the string

A B A B A C

as a pattern, and let's list all its prefixes:

0 /the empty string/
1 A
2 A B
3 A B A
4 A B A B

5 A B A B A
6 A B A B A C

Let us now consider for each such listed string (prefix) the longest proper suffix (a suffix different from the string itself), which is at the same time a prefix of it:

0 /the empty string/
1 /the empty string/
2 /the empty string/
3 A
4 A B
5 A B A
6 /the empty string/

It's easy to see that if we have at some point a partial match up to say the prefix (A B A B A) we also have a partial match up to the prefixes (A B A), and (A) - which are both prefixes of the initial string and suffix/prefixes of the current match. Depending on the next "incoming" character from the text, three cases arise:

1. The next character is C. We can "expand" the match at the level of the prefix (A B A B A). In this particular case this leads to a full match and we just notice this fact.
2. The next character is B. The partial match for the prefix (A B A B A) cannot be "expanded". The best we can do is to return to the largest different partial match we have so far - the prefix (A B A) and try to "expand" it. Now B "fits" so we continue with the next character from the text and our current "best" partial match will become the string (A B A B) from our "list of prefixes".
3. The "incoming" character is, for example, D. The "journey" back to (A B A) is obviously insufficient to "expand" the match. In this case we have to go further back to the second largest partial match (the second largest proper suffix of the initial match that is at the same time a prefix of it) - that is (A) and finally to the empty string (the third largest proper suffix in our case). Since it turns out that there is no way to "expand" even the empty string using the character D, we skip D and go to the next character from the text. But now our "best" partial match so far will be the empty string.

In order to build the KMP automaton (or the so called KMP "failure function") we have to initialize an integer array F[]. The indexes (from 0 to m - the length of the pattern) represent the numbers under which the consecutive prefixes of the pattern are listed in our "list of prefixes" above. Under each index is a "pointer" - that identifies the index of the longest proper suffix, which is at the same time a prefix of the given string (or in other words F[i] is the index of next best partial match for the string under index i). In our case (the string A B A B A C) the array F[] will look as follows:

F[0] = 0
F[1] = 0
F[2] = 0
F[3] = 1
F[4] = 2
F[5] = 3
F[6] = 0

Notice that after initialization F[i] contains information not only about the largest next partial match for the string under index i but also about every partial match of it. F[i] is the first best partial match, F[F[i]] - is the second best, F[F[F[i]]] - the third, and so on. Using this information we can calculate F[i] if we know the values F[k] for all k < i. The best next partial match of string i will be the largest partial match of string i - 1 whose character that "expands" it is equal to the last character of string i. So all we need to do is to check every partial match of string i - 1 in descending order of length and see if the last character of string i "expands" the match at this level. If no partial match can be "expanded" than F[i] is the empty string. Otherwise F[i] is the largest "expanded" partial match (after its "expansion").

In terms of pseudocode the initialization of the array F[] (the "failure function") may look like this:

```
// Pay attention!
// the prefix under index i in the table above is
// is the string from pattern[0] to pattern[i - 1]
// inclusive, so the last character of the string under
// index i is pattern[i - 1]

function build_failure_function(pattern[])
{
  // let m be the length of the pattern
```

```
  F[0] = F[1] = 0; // always true

  for(i = 2; i <= m; i++) {
    // j is the index of the largest next partial match
    // (the largest suffix/prefix) of the string under
    // index i - 1
    j = F[i - 1];
    for( ; ; ) {
      // check to see if the last character of string i -
      // - pattern[i - 1] "expands" the current "candidate"
      // best partial match - the prefix under index j
      if(pattern[j] == pattern[i - 1]) {
        F[i] = j + 1; break;
      }
      // if we cannot "expand" even the empty string
      if(j == 0) { F[i] = 0; break; }
      // else go to the next best "candidate" partial match
      j = F[j];
    }
  }
}
```

The automaton consists of the initialized array F[] ("internal rules") and a pointer to the index of the prefix of the pattern that is the best (largest) partial match that ends at the current position in the text ("current state"). The use of the automaton is almost identical to what we did in order to build the "failure function". We take the next character from the text and try to "expand" the current partial match. If we fail, we go to the next best partial match of the current partial match and so on. According to the index where this procedure leads us, the "current state" of the automaton is changed. If we are unable to "expand" even the empty string we just skip this character, go to the next one in the text, and the "current state" becomes zero.

```
function Knuth_Morris_Pratt(text[], pattern[])
{
  // let n be the size of the text, m the
  // size of the pattern, and F[] - the
  // "failure function"

  build_failure_function(pattern[]);

  i = 0; // the initial state of the automaton is
         // the empty string

  j = 0; // the first character of the text

  for( ; ; ) {
    if(j == n) break; // we reached the end of the text

    // if the current character of the text "expands" the
    // current match
    if(text[j] == pattern[i]) {
      i++; // change the state of the automaton
      j++; // get the next character from the text
      if(i == m) // match found
    }

    // if the current state is not zero (we have not
    // reached the empty string yet) we try to
    // "expand" the next best (largest) match
    else if(i > 0) i = F[i];

    // if we reached the empty string and failed to
    // "expand" even it; we go to the next
    // character from the text, the state of the
    // automaton remains zero
    else j++;
  }
}
```

Many problems in programming contests focus more on the properties of KMP's "failure function," rather than on its use for string matching. An example is: given a string (a quite long one), find all its proper suffixes that are also prefixes of it. All we have to do is just to calculate the "failure function" of the given string and using the information stored in it to print the answer.

A typical problem seen quite often is: given a string find its shortest substring, such that the concatenation of one or more copies of it results in the original string. Again the problem can be reduced to the properties of the failure function. Let's consider the string

A B A B A B

and all its proper suffix/prefixes in descending order:

1 A B A B
2 A B
3 /the empty string/

Every such suffix/prefix uniquely defines a string, which after being "inserted" in front of the given suffix/prefix gives the initial string. In our case:

1 A B
2 A B A B
3 A B A B A B

Every such "augmenting" string is a potential "candidate" for a string, the concatenation of several copies of which results in the initial string. This follows from the fact that it is not only a prefix of the initial string but also a prefix of the suffix/prefix it "augments". But that means that now the suffix/prefix contains at least two copies of the "augmenting" string as a prefix (since it's also a prefix of the initial string) and so on. Of course if the suffix/prefix under question is long enough. In other words, the length of a successful "candidate" must divide with no remainder the length of the initial string.

So all we have to do in order to solve the given problem is to iterate through all proper suffixes/prefixes of the initial string in descending order. This is just what the "failure function" is designed for. We iterate until we find an "augmenting" string of the desired length (its length divides with no remainder the length of the initial string) or get to the empty string, in which case the "augmenting" string that meets the above requirement will be the initial string itself.

Rabin-Karp and Knuth-Morris-Pratt at TopCoder
In the problem types mentioned above, we are dealing with relatively "pure" forms of RK, KMP and the techniques that are the essence of these algorithms. While you're unlikely to encounter these pure situations in a TopCoder SRM, the drive towards ever more challenging TopCoder problems can lead to situations where these algorithms appear as one level in complex, "multilayer" problems. The specific input size limitations favor this trend, since we will not be presented as input with multimillion character strings, but rather with a "generator", which may be by itself algorithmic in nature. A good example is "InfiniteSoup," Division 1 - Level Three, SRM 286.

With this article, we'll revisit the so-called "max-flow" problem, with the goal of making some practical analysis of the most famous augmenting path algorithms. We will discuss several algorithms with different complexity from $O(nm^2)$ to O(nmlogU) and reveal the most efficient one in practice. As we will see, theoretical complexity is not a good indicator of the actual value of an algorithm.

The article is targeted to the readers who are familiar with the basics of network flow theory. If not, I'll refer them to check out [1], [2] or algorithm tutorial [5].

In the first section we remind some necessary definitions and statements of the maximum flow theory. Other sections discuss the augmenting path algorithms themselves. The last section shows results of a practical analysis and highlights the best in practice algorithm. Also we give a simple implementation of one of the algorithms.

Statement of the Problem

Suppose we have a directed network G = (V, E) defined by a set V of nodes (or vertexes) and a set E of arcs (or edges). Each arc (i,j) in E has an associated nonnegative capacity u_{ij}. Also we distinguish two special nodes in G: a *source* node s and a *sink* node t. For each i in V we denote by E(i) all the arcs emanating from node i. Let U = max u_{ij} by (i,j) in E. Let us also denote the number of vertexes by n and the number of edges by m.

We wish to find the maximum flow from the source node s to the sink node t that satisfies the arc capacities and mass balance constraints at all nodes. Representing the flow on arc (i,j) in E by x_{ij} we can obtain the optimization model for the maximum flow problem:

$$\text{Maximize } f(x) = \sum_{(i,j)\in E(s)} x_{ij}$$

$$\text{subject to } \sum_{\{j:\,(i,j)\in E\}} x_{ij} - \sum_{\{j:\,(j,i)\in E\}} x_{ji} = 0 \quad \forall i \in V \setminus \{s,t\}$$

$$0 \leqslant x_{ij} \leqslant u_{ij} \quad \forall (i,j) \in E$$

Vector (x_{ij}) which satisfies all constraints is called *a feasible solution* or, *a flow* (it is not necessary maximal). Given a flow x we are able to construct the residual network with respect to this flow according to the following intuitive idea. Suppose that an edge (i,j) in E carries x_{ij} units of flow. We define the residual capacity of the edge (i,j) as $r_{ij} = u_{ij} - x_{ij}$. This means that we can send an additional r_{ij} units of flow from vertex i to vertex j. We can also cancel the existing flow x_{ij} on the arc if we send up x_{ij} units of flow from j to i over the arc (i,j).

So, given a feasible flow x we define the residual network with respect to the flow x as follows. Suppose we have a network G = (V, E). A feasible solution x engenders a new (residual) network, which we define by $G_x = (V, E_x)$, where E_x is a set of residual edges corresponding to the feasible solution x.

What is E_x? We replace each arc (i,j) in E by two arcs (i,j), (j,i): the arc (i,j) has (residual) capacity $r_{ij} = u_{ij} - x_{ij}$, and the arc (j,i) has (residual) capacity $r_{ji}=x_{ij}$. Then we construct the set E_x from the new edges with a positive residual capacity.

Augmenting Path Algorithms as a whole

In this section we describe one method on which all augmenting path algorithms are being based. This method was developed by Ford and Fulkerson in 1956 [3]. We start with some important definitions.

Augmenting path is a directed path from a source node s to a sink node t in the residual network. The residual capacity of an augmenting path is the minimum residual capacity of any arc in the path. Obviously, we can send additional flow from the source to the sink along an augmenting path.

All augmenting path algorithms are being constructed on the following basic idea known as augmenting path theorem:

***Theorem 1 (Augmenting Path Theorem)**. A flow x* is a maximum flow if and only if the residual network Gx* contains no augmenting path.*

According to the theorem we obtain a method of finding a maximal flow. The method proceeds by identifying augmenting paths and augmenting flows on these paths until the network contains no such path. All algorithms that we wish to discuss differ only in the way of finding augmenting paths.

We consider the maximum flow problem subject to the following assumptions.

Assumption 1. *The network is directed.*

Assumption 2. *All capacities are nonnegative integers.*

This assumption is not necessary for some algorithms, but the algorithms whose complexity bounds involve U assume the integrality of the data.

Assumption 3. *The problem has a bounded optimal solution.*

This assumption in particular means that there are no uncapacitated paths from the source to the sink.

Assumption 4. *The network does not contain parallel arcs.*

This assumption imposes no loss of generality, because one can summarize capacities of all parallel arcs.

As to why these assumptions are correct we leave the proof to the reader.

It is easy to determine that the method described above works correctly. Under assumption 2, on each augmenting step we increase the flow value by at least one unit. We (usually) start with zero flow. The maximum flow value is bounded from above, according to assumption 3. This reasoning implies the finiteness of the method.

With those preparations behind us, we are ready to begin discussing the algorithms.

Shortest Augmenting Path Algorithm, $O(n^2m)$

In 1972 Edmonds and Karp -- and, in 1970, Dinic -- independently proved that if each augmenting path is shortest one, the algorithm will perform O(nm) augmentation steps. The shortest path (length of each edge is equal to one) can be found with the help of breadth-first search (BFS) algorithm [2], [6]. Shortest Augmenting Path Algorithm is well known and widely discussed in many books and articles, including [5], which is why we will not describe it in great detail. Let's review the idea using a kind of pseudo-code:

```
SHORTEST—AUGMENTING—PATH
1  x ← 0
2  while in G_x exists path s ⇝ t
3        do find a shortest augmenting path P
4           δ ← min{r_ij : (i, j) ∈ P}
5           augment δ units of flow along P
6           update G_x
7  return x
```

In line 5, current flow x is being increased by some positive amount.

The algorithm was said to perform O(nm) steps of finding an augmenting path. Using BFS, which requires O(m) operation in the worst case, one can obtain $O(nm^2)$ complexity of the algorithm itself. If m ~ n2 then one must use BFS procedure $O(n^3)$ times in worst case. There are some networks on which this numbers of augmentation steps is being achieved. We will show one simple example below.

Improved Shortest Augmenting Path Algorithm, $O(n^2m)$

As mentioned earlier, the natural approach for finding any shortest augmenting path would be to look for paths by performing a breadth-first search in the residual network. It requires O(m) operations in the worst case and imposes $O(nm^2)$ complexity of the maximum flow algorithm. Ahuja and Orlin improved the shortest augmenting path algorithm in 1987 [1]. They exploited the fact that the minimum distance from any node i to the sink node t is monotonically nondecreasing over all augmentations and reduced the average time per augmentation to O(n). The improved version of the augmenting path algorithm, then, runs in $O(n^2m)$ time. We can now start discussing it according to [1].

Definition 1. *Distance function d: V_ Z+ with respect to the residual capacities rij is a function from the set of nodes to nonnegative integers. Let's say that distance function is valid if it is satisfies the following conditions:*

- *d(t)=0;*
- *d(i) ≤ d(j) + 1, for every (i,j) in E with rij>0.*

Informally (and it is easy to prove), valid distance label of node i, represented by d(i), is a lower bound on the length of the shortest path from i to t in the residual network Gx. We call distance function *exact* if each i in V d(i) equals the length of the shortest path from i to t in the residual network. It is also easy to prove that if d(s) ≥ n then the residual network contains no path from the source to the sink.

An arc (i,j) in E is called admissible if d(i) = d(j) + 1. We call other arcs *inadmissible*. If a path from s to t consists of admissible arcs then the path is admissible. Evidently, an admissible path is the shortest path from the source to the sink. As far as every arc in an admissible path satisfies condition $r_{ij}>0$, the path is augmenting.

So, the improved shortest augmenting path algorithm consists of four steps (procedures): *main cycle, advance, retreat* and *augment*. The algorithm maintains a *partial admissible path*, i.e., a path from s to some node i, consisting of admissible arcs. It performs *advance* or *retreat* steps from the last node of the partial admissible path (such node is called *current node*). If there is some admissible arc (i,j) from current node i, then the algorithm performs the *advance* step and adds the arc to the partial admissible path. Otherwise, it performs the *retreat* step, which increases distance label of i and backtracks by one arc.

If the partial admissible path reaches the sink, we perform an augmentation. The algorithm stops when d(s) ≥ n. Let's describe these steps in pseudo-code [1]. We denoted residual (with respect to flow x) arcs emanating from node i by $E_x(i)$. More formally, $E_x(i) = \{ (i,j) \text{ in } E(i): r_{ij} > 0 \}$.

```
algorithm IMPROVED—SHORTEST—AUGMENTING—PATH
 1  x ← 0
 2  Perform a (reverse) breadth-first search of the residual network
    starting with the sink node to compute exact distance labels d(i)
 3  i ← s
 4  while d(s) < n
 5        do if G_x contains an admissible arc (i, j) ∈ E_x(i)
 6              then ADVANCE(i)
 7                   if i = t                ▷ We found an augmenting path
 8                     then AUGMENT
 9                          i ← s            ▷ Begin finding next path
10              else  RETREAT(i)     ▷ No admissible arc emanating from i
11  return x

procedure ADVANCE(i)
1  let (i, j) be an admissible arc in E_x(i)
2  π(j) ← i     ▷ We maintain the predecessor list
3  i ← j        ▷ Make node j as current node

procedure RETREAT(i)
1  d(i) ← 1 + min{d(j) : (i, j) ∈ E_x(i)}  ▷ This operation is called relabel
2  if i ≠ s
3     then i ← π(i)   ▷ Backtrack

procedure AUGMENT
1  using the predecessor indices π identity an augmenting path P
2  δ ← min{r_ij : (i, j) ∈ P}
3  augment δ units of flow along P
4  update G_x (or, E_x)
```

In line 1 of *retreat* procedure if $E_x(i)$ is empty, then suppose d(i) equals n.

Ahuja and Orlin suggest the following data structure for this algorithm [1]. We maintain the arc list E(i) which contains all the arcs emanating from node i. We arrange the arcs in this list in any fixed order. Each node i has a *current arc*, which is an arc in E(i) and is the next candidate for admissibility testing. Initially, the current arc of node i is the first arc in E(i). In line 5 the algorithm tests whether the node's current arc is admissible. If not, it designates the next arc in the list as the current arc. The algorithm repeats this process until either it finds an admissible arc or reaches the end of the arc list. In the latter case the algorithm declares that E(i) contains no admissible arc; it again designates the first arc in E(i) as the current arc of node i and performs the *relabel* operation by calling the *retreat* procedure (line 10).

Now we outline a proof that the algorithm runs in $O(n^2m)$ time.

Lemma 1. *The algorithm maintains distance labels at each step. Moreover, each relabel (or, retreat) step strictly increases the distance label of a node.*

Sketch to proof. Perform induction on the number of *relabel* operation and augmentations.

Lemma 2. *Distance label of each node increases at most n times. Consecutively, relabel operation performs at most n^2 times.*

Proof. This lemma is consequence of lemma 1 and the fact that if d(s) ≥ n then the residual network contains no augmenting path.

Since the improved shortest augmenting path algorithm makes augmentations along the shortest paths (like unimproved one), the total number of augmentations is the same O(nm). Each *retreat* step relabels a node, that is why number of *retreat* steps is $O(n^2)$ (according to lemma 2). Time to perform *retreat/relabel* steps is $O(n \sum_{i \text{ in } V} |E(i)|) = O(nm)$. Since one augmentation requires O(n) time, total augmentation time is $O(n^2m)$. The total time of *advance* steps is bounded by the augmentation time plus the *retreat/relabel* time and it is again $O(n^2m)$. We obtain the following result:

Theorem 2. *The improved shortest augmenting path algorithm runs in $O(n^2m)$ time.*

Ahuja and Orlin [1] suggest one very useful practical improvement of the algorithm. Since the algorithm performs many useless relabel operations while the maximum flow has been found, it will be better to give an additional criteria of terminating. Let's introduce (n+1)-dimensional additional array, *numbs*, whose indices vary from 0 to n. The value *numbs(k)* is the number of nodes whose distance label equals *k*. The algorithm initializes this array while computing the initial distance labels using BFS. At this point, the positive entries in the array *numbs* are consecutive (i.e., *numbs(0), numbs(1), ..., numbs(l)* will be positive up to some index l and the remaining entries will all be zero).

When the algorithm increases a distance label of a node from x to y, it subtracts 1 from *numbs(x)*, adds 1 to *numbs(y)* and checks whether *numbs(x) = 0*. If it does equal 0, the algorithm terminates.

This approach is some kind of heuristic, but it is really good in practice. As to why this approach works we leave the proof to the reader (*hint*: show that the nodes i with d(i) > x and nodes j with d(j) < x engender a cut and use maximum-flow-minimum-cut theorem).

Comparison of Improved and Unimproved versions
In this section we identify the worst case for both shortest augmenting path algorithms with the purpose of comparing their running times.

In the worst case both improved and unimproved algorithms will perform $O(n^3)$ augmentations, if m ~ n2. Norman Zadeh [4] developed some examples on which this running time is based. Using his ideas we compose a somewhat simpler network on which the algorithms have to perform $O(n^3)$ augmentations and which is not dependent on a choice of next path.

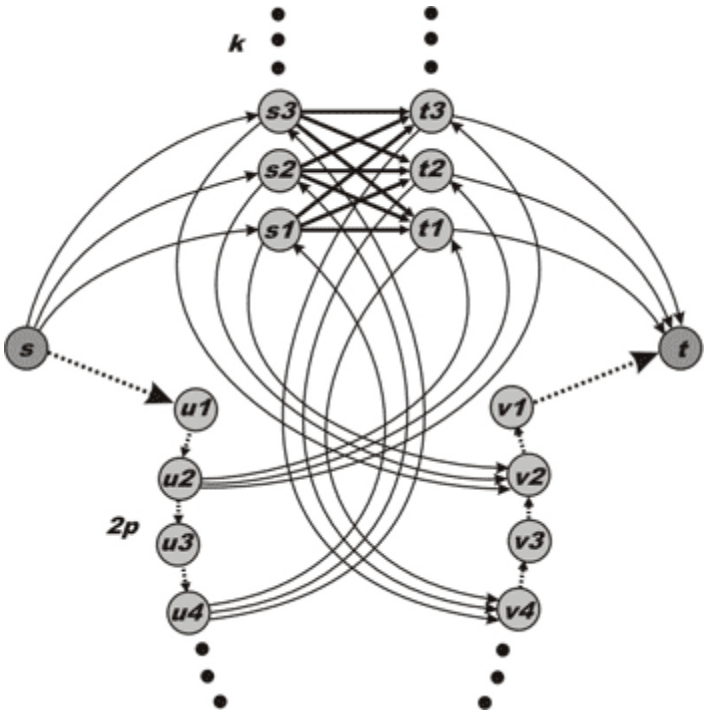


Figure 1. Worst case example for the shortest augmenting path algorithm.

All vertexes except s and t are divided into four subsets: S={s1,...,sk}, T={t1,...,tk}, U={u1,...,u2p} and V={v1,...,v2p}. Both sets S and T contain k nodes while both sets U and V contain 2p nodes. k and p are fixed integers. Each bold arc (connecting S and T) has unit capacity. Each dotted arc has an infinite capacity. Other arcs (which are solid and not straight) have capacity k.

First, the shortest augmenting path algorithm has to augment flow k^2 time along paths (s, S, T, t) which have length equal to 3. The capacities of these paths are unit. After that the residual network will contain reversal arcs (T, S) and the algorithm will chose another k^2 augmenting paths (*s, u1, u2, T, S, v2, v1, t*) of length 7. Then the algorithm will have to choose paths (*s, u1, u2, u3, u4, S, T, v4, v3, v2, v1, t*) of length 11 and so on...

Now let's calculate the parameters of our network. The number of vertexes is n = 2k + 4p + 2. The number of edges is m = k^2 + 2pk + 2k + 4p. As it easy to see, the number of augmentations is a = k2 (p+1).

Consider that p = k - 1. In this case n = 6k - 2 and a = k^3. So, one can verify that a ~ n^3 / 216. In [4] Zadeh presents examples of networks that require n^3 / 27 and n^3 / 12 augmentations, but these examples are dependent on a choice of the shortest path.

We made 5 worst-case tests with 100, 148, 202, 250 and 298 vertexes and compared the running times of the improved version of the algorithm against the unimproved one. As you can see on figure 2, the improved algorithm is much faster. On the network with 298 nodes it works 23 times faster. Practice analysis shows us that, in general, the improved algorithm works n / 14 times faster.

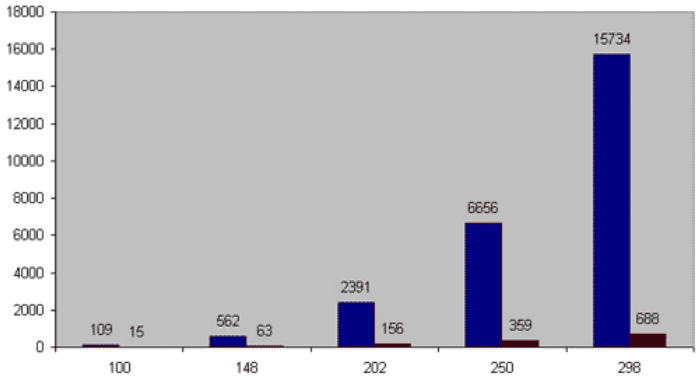


Figure 2. X-axis is the number of nodes. Y-axis is working time in milliseconds.
Blue colour indicates the shortest augmenting path algorithm and red does it improved version.

However, our comparison in not definitive, because we used only one kind of networks. We just wanted to justify that the $O(n^2m)$ algorithm works O(n) times faster than the $O(nm^2)$ on a dense network. A more revealing comparison is waiting for us at the end of the article.

Maximum Capacity Path Algorithm, *$O(n^2m\log nU)$ / $O(m^2 \log nU \log n)$ / $O(m^2 \log nU \log U)$*
In 1972 Edmonds and Karp developed another way to find an augmenting path. At each step they tried to increase the flow with the maximum possible amount. Another name of this algorithm is "gradient modification of the Ford-Fulkerson method." Instead of using BFS to identify a shortest path, this modification uses Dijkstra's algorithm to establish a path with the maximal possible capacity. After augmentation, the algorithm finds another such path in the residual network, augments flow along it, and repeats these steps until the flow is maximal.

```
MAXIMUM—CAPACITY—PATH
1  x ← 0
2  while in G_x exists path s ⇝ t
3        do find an augmenting path P with the maximum capacity
4           δ ← min{r_ij : (i, j) ∈ P}
5           augment δ units of flow along P
6           update G_x
7  return x
```

There's no doubt that the algorithm is correct in case of integral capacity. However, there are tests with non-integral arc's capacities on which the algorithm may fail to terminate.

Let's get the algorithm's running time bound by starting with one lemma. To understand the proof one should remember that the value of any flow is less than or equal to the capacity of any cut in a network, or read this proof in [1], [2]. Let's denote capacity of a cut (S,T) by c(S,T).

Lemma 3. *Let F be the maximum flow's value, then G contains augmenting path with capacity not less than F/m.*

Proof. Suppose G contains no such path. Let's construct a set E'={ (i,j) in E: $u_{ij} \geq F/m$ }. Consider network G' = (V, E') which has no path from s to t. Let S be a set of nodes obtainable from s in G and T = V \ S. Evidently, (S, T) is a cut and $c(S, T) \geq F$. But cut (S, T) intersects only those edges (i,j) in E which have $u_{ij} < F/m$. So, it is clear that

$$c(S,T) < (F/m) \_ m = F,$$

and we got a contradiction with the fact that $c(S,T) \geq F$.

Theorem 3. *The maximum capacity path algorithm performs O(m log (nU)) augmentations.*

Sketch to proof. Suppose that the algorithm terminates after k augmentations. Let's denote by f_1 the capacity of the first found augmentation path, by f_2 the capacity of the second one, and so on. f_k will be the capacity of the latter k-th augmenting path.

Consider, $F_i = f_1 + f_2 + ... + f_i$. Let F* be the maximum flow's value. Under lemma 3 one can justify that

$$f_i \geq (F^*-F_{i-1}) / m.$$

Now we can estimate the difference between the value of the maximal flow and the flow after i consecutive augmentations:

$$F^* - F_i = F^* - F_{i-1} - f_i \leq F^* - F_{i-1} - (F^* - F_{i-1}) / m = (1 - 1 / m) (F^* - F_{i-1}) \leq ... \leq (1 - 1 / m)^i \_ F^*$$

We have to find such an integer i, which gives $(1 - 1 / m)^i \_ F^* < 1$. One can check that

$$i \log_{m / (m+1)} F^* = O(m \_ \log F^*) = O(m \_ \log(nU))$$

And the latter inequality proves the theorem.

To find a path with the maximal capacity we use Dijkstra's algorithm, which incurs additional expense at every iteration. Since a simple realization of Dijkstras's algorithm [2] incurs $O(n^2)$ complexity, the total running time of the maximum capacity path algorithm is $O(n^2 m \log(nU))$.

Using a heap implementation of Dijkstra's algorithm for sparse network [7] with running time O(mlogn), one can obtain an $O(m^2 \log n \log(nU))$ algorithm for finding the maximum flow. It seems to be better that the improved Edmonds-Karp algorithm. However, this estimate is very deceptive.

There is another variant to find the maximum capacity path. One can use binary search to establish such a path. Let's start by finding the maximum capacity path on piece [0,U]. If there is some path with capacity U/2, then we continue finding the path on piece [U/2, U]; otherwise, we try to find the path on [0,U/2-1]. This approach incurs additional O(mlogU) expense and gives $O(m^2 \log(nU) \log U)$ time bound to the maximum flow algorithm. However, it works really poorly in practice.

Capacity Scaling Algorithm, *O(m^2logU)*

In 1985 Gabow described the so-called "bit-scaling" algorithm. The similar capacity scaling algorithm described in this section is due to Ahuja and Orlin [1].

Informally, the main idea of the algorithm is to augment the flow along paths with sufficient large capacities, instead of augmenting along maximal capacities. More formally, let's introduce a parameter *Delta*. First, Delta is quite a large number that, for instance, equals U. The algorithm tries to find an augmenting path with capacity not less that Delta, then augments flow along this path and repeats this procedure while any such Delta-path exists in the residual network.

The algorithm either establishes a maximum flow or reduces Delta by a factor of 2 and continues finding paths and augmenting flow with the new Delta. The phase of the algorithm that augments flow along paths with capacities at least Delta is called *Delta-scaling phase* or, *Delta-phase*. Delta is an integral value and, evidently, algorithm performs O(logU) Delta-phases. When Delta is equal to 1 there is no difference between the capacity scaling algorithm and the Edmonds-Karp algorithm, which is why the algorithm works correctly.

```
CAPACITY—SCALING
1  x ← 0
2  Δ ← 2^⌊log2 U⌋
3  while Δ > 0
4        do while in G_x exists path s ⇝ t with the capacity of at least Δ
5              do find an augmenting path P with the capacity of at least Δ
6                 δ ← min{r_ij : (i, j) ∈ P}
7                 augment δ units of flow along P
8                 update G_x
9           Δ ← Δ/2          ▷ Integer division
10 return x
```

We can obtain a path with the capacity of at least Delta fairly easily - in O(m) time (by using BFS). At the first phase we can set Delta to equal either U or the largest power of 2 that doesn't exceed U.

The proof of the following lemma is left to the reader.

Lemma 4. *At every Delta-phase the algorithm performs O(m) augmentations in worst case.*

Sketch to proof. Use the induction by Delta to justify that the minimum cut at each Delta-scaling phase less that 2m Delta.

Applying lemma 4 yields the following result:

Theorem 4. *Running time of the capacity scaling algorithm is O(m^2logU).*

Keep in mind that there is no difference between using breadth-first search and depth-first search when finding an augmenting path. However, in practice, there is a big difference, and we will see it.

Improved Capacity Scaling Algorithm, O(nmlogU)
In the previous section we described an O(m^2logU) algorithm for finding the maximum flow. We are going to improve the running time of this algorithm to O(nmlogU) [1].

Now let's look at each Delta-phase independently. Recall from the preceding section that a Delta-scaling phase contains O(m) augmentations. Now we use the similar technique at each Delta-phase that we used when describing the improved variant of the shortest augmenting path algorithm. At every phase we have to find the "maximum" flow by using only paths with capacities equal to at least Delta. The complexity analysis of the improved shortest augmenting path algorithm implies that if the algorithm is guaranteed to perform O(m) augmentations, it would run in O(nm) time because the time for augmentations reduces from O(n^2m) to O(nm) and all other operations, as before, require O(nm) time. These reasoning instantly yield a bound of O(nmlogU) on the running time of the improved capacity scaling algorithm.

Unfortunately, this improvement hardly decreases the running time of the algorithm in practice.

Practical Analysis and Comparison
Now let's have some fun. In this section we compare all described algorithms from practical point of view. With this purpose I have made some test cases with the help of [8] and divided them into three groups by density. The first group of tests is made of networks with m ≤ n1.4 - some kind of sparse networks. The second one consists of middle density tests with n1.6 ≤ m ≤ n1.7. And the last group represents some kinds of almost full graphs (including full acyclic networks) with m ≥ n1.85.

I have simple implementations of all algorithms described before. I realized these algorithms without any kind of tricks and with no preference towards any of them. All implementations use adjacency list for representing a network.

Let's start with the first group of tests. These are 564 sparse networks with number of vertexes limited by 2000 (otherwise, all algorithms work too fast). All working times are given in milliseconds.

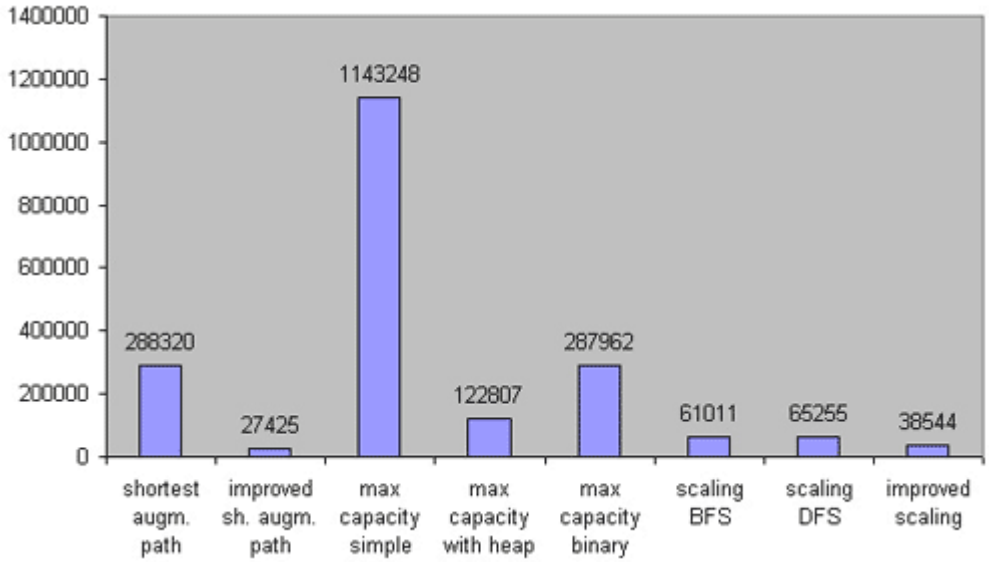


Figure 3. Comparison on sparse networks. 564 test cases. $m \le n1.4$.

As you can see, it was a big mistake to try Dijkstra's without heap implementation of the maximum capacity path algorithm on sparse networks (and it's not surprising); however, its heap implementation works rather faster than expected. Both the capacity scaling algorithms (with using DFS and BFS) work in approximately the same time, while the improved implementation is almost 2 times faster. Surprisingly, the improved shortest path algorithm turned out to be the fastest on sparse networks.

Now let's look at the second group of test cases. It is made of 184 tests with middle density. All networks are limited to 400 nodes.

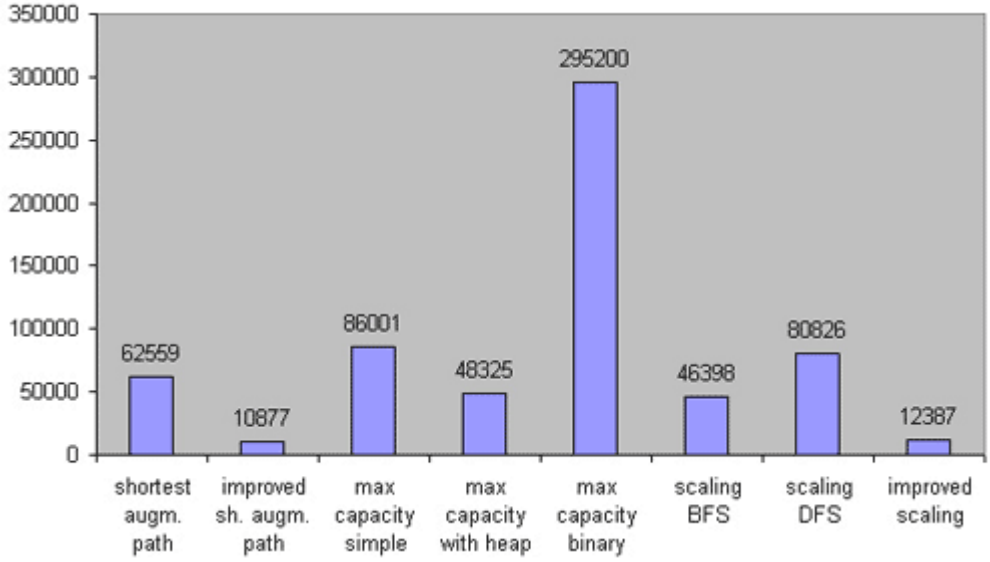


Figure 4. Comparison on middle density networks. 184 test cases. $n1.6 \le m \le n1.7$.

On the middle density networks the binary search implementation of the maximum capacity path algorithm leaves much to be desired, but the heap implementation still works faster than the simple (without heap) one. The BFS realization of the capacity scaling algorithm is faster than the DFS one. The improved scaling algorithm and the improved shortest augmenting path algorithm are both very good in this case.

It is very interesting to see how these algorithms run on dense networks. Let's take a look -- the third group is made up of 200 dense networks limited by 400 vertexes.

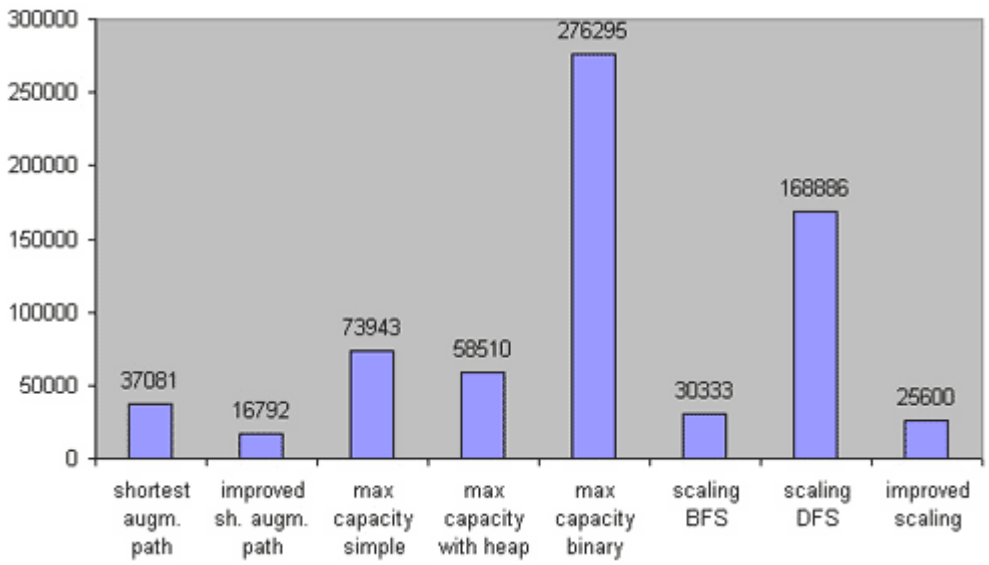


Figure 5. Comparison on dense networks. 200 test cases. $m \geq n1.85$.

Now we see the difference between the BFS and DFS versions of the capacity scaling algorithm. It is interesting that the improved realization works in approximately the same time as the unimproved one. Unexpectedly, the Dijkstra's with heap implementation of the maximum capacity path algorithm turned out to be faster than one without heap.

Without any doubt, the improved implementation of Edmonds-Karp algorithm wins the game. Second place is taken by the improved scaling capacity algorithm. And the scaling capacity with BFS got bronze.

As to maximum capacity path, it is better to use one variant with heap; on sparse networks it gives very good results. Other algorithms are really only good for theoretical interest.

As you can see, the O(nmlogU) algorithm isn't so fast. It is even slower than the $O(n^2m)$ algorithm. The $O(nm^2)$ algorithm (it is the most popular) has worse time bounds, but it works much faster than most of the other algorithms with better time bounds.

My recommendation: Always use the scaling capacity path algorithm with BFS, because it is very easy to implement. The improved shortest augmenting path algorithm is rather easy, too, but you need to be very careful to write the program correctly. During the contest it is very easy to miss a bug.

I would like to finish the article with the full implementation of the improved shortest augmenting path algorithm. To maintain a network I use the adjacency matrix with purpose to providing best understanding of the algorithm. It is not the same realization what was used during our practical analysis. With the "help" of the matrix it works a little slower than one that uses adjacency list. However, it works faster on dense networks, and it is up to the reader which data structure is best for them.

```
#include <stdio.h>

#define N 2007  // Number of nodes
#define oo 1000000000 // Infinity

// Nodes, Arcs, the source node and the sink node
int n, m, source, sink;

// Matrixes for maintaining
// Graph and Flow
int G[N][N], F[N][N];

int pi[N];  // predecessor list
int CurrentNode[N]; // Current edge for each node

int queue[N];  // Queue for reverse BFS

int d[N];  // Distance function
int numbs[N];  // numbs[k] is the number of nodes i with d[i]==k
```

```
// Reverse breadth-first search
// to establish distance function d
int rev_BFS()  {
  int i, j, head(0), tail(0);

  // Initially, all d[i]=n
  for(i = 1; i <= n; i++)
    numbs[ d[i] = n ] ++;

  // Start from the sink
  numbs[n]--;
  d[sink] = 0;
  numbs[0]++;

  queue[ ++tail ] = sink;

  // While queue is not empty
  while( head != tail )  {
    i = queue[++head];  // Get the next node

    // Check all adjacent nodes
    for(j = 1; j <= n; j++)  {

      // If it was reached before or there is no edge
      // then continue
      if(d[j] < n || G[j][i] == 0)  continue;

      // j is reached first time
      // put it into queue
      queue[ ++tail ] = j;

      // Update distance function
      numbs[n]--;
      d[j] = d[i] + 1;
      numbs[d[j]]++;

    }
  }

  return 0;
}

// Augmenting the flow using predecessor list pi[]
int Augment()  {
  int i, j, tmp, width(oo);

  // Find the capacity of the path
  for(i = sink, j = pi[i]; i != source; i = j, j = pi[j])  {
    tmp = G[j][i];
    if(tmp < width)  width = tmp;
  }

  // Augmentation itself
  for(i = sink, j = pi[i]; i != source; i = j, j = pi[j])  {
    G[j][i] -= width;  F[j][i] += width;
    G[i][j] += width;  F[i][j] -= width;
  }

  return width;
}

// Relabel and backtrack
int Retreat(int &i)  {
  int tmp;
  int j, mind(n-1);

  // Check all adjacent edges
  // to find nearest
```

```
  for(j=1; j <= n; j++)
    // If there is an arc
    // and j is "nearer"
    if(G[i][j] > 0 && d[j] < mind)
      mind = d[j];

  tmp = d[i];  // Save previous distance

  // Relabel procedure itself
  numbs[d[i]]--;
  d[i] = 1 + mind;
  numbs[d[i]]++;

  // Backtrack, if possible (i is not a local variable! )
  if( i != source )  i = pi[i];

  // If numbs[ tmp ] is zero, algorithm will stop
  return numbs[ tmp ];
}

// Main procedure
int find_max_flow()  {
  int flow(0), i, j;

  rev_BFS();  // Establish exact distance function

  // For each node current arc is the first arc
  for(i=1; i<=n; i++)  CurrentNode[i] = 1;

  // Begin searching from the source
  i = source;

  // The main cycle (while the source is not "far" from the sink)
  for( ; d[source] < n ; )  {

    // Start searching an admissible arc from the current arc
    for(j = CurrentNode[i]; j <= n; j++)
      // If the arc exists in the residual network
      // and if it is an admissible
      if( G[i][j] > 0 && d[i] == d[j] + 1 )
        // Then finish searhing
        break;

    // If the admissible arc is found
    if( j <= n )  {
      CurrentNode[i] = j;  // Mark the arc as "current"
      pi[j] = i;  //  j is reachable from i
      i = j;  // Go forward

      // If we found an augmenting path
      if( i == sink )  {
        flow += Augment();  // Augment the flow
        i = source;  // Begin from the source again
      }
    }
    // If no an admissible arc found
    else  {
      CurrentNode[i] = 1;  // Current arc is the first arc again

      // If numbs[ d[i] ] == 0  then the flow is the maximal
      if( Retreat(i) == 0 )
        break;

    }

  } // End of the main cycle

  // We return flow value
  return flow;
```

```
}


// The main function
// Graph is represented in input as triples <from, to, capacity>

// No comments here
int main()  {
  int i, p, q, r;

  scanf("%d %d %d %d", &n, &m, &source, &sink);

  for(i = 0; i < m; i++)  {
    scanf("%d %d %d", &p, &q, &r);
    G[p][q] += r;
  }

  printf("%d", find_max_flow());

  return 0;
}
```

Introduction

Counting the objects that satisfy some criteria is a very common task in both TopCoder problems and in real-life situations. The myriad ways of counting the number of elements in a set is one of the main tasks in combinatorics, and I'll try to describe some basic aspects of it in this tutorial. These methods are used in a range of applications, from discrete math and probability theory to statistics, physics, biology, and more.

Combinatorial primitives

Let's begin with a quick overview of the basic rules and objects that we will reference later.

The rule of sum

$$A \cap B = \varnothing \Rightarrow |A| + |B| = |A \cup B|$$

The rule of product

$$|A| \cdot |B| = |A \times B|$$

For example, if we have three towns -- A, B and C -- and there are 3 roads from A to B and 5 roads from B to C, then we can get from A to C through B in 3*5=15 different ways.

These rules can be used for a finite collections of sets.

Permutation without repetition

When we choose *k* objects from *n*-element set in such a way that the order matters and each object can be chosen only once:

$$(n)_k = P_k^n = \frac{n!}{(n-k)!}$$

For example, suppose we are planning the next 3 contests and we have a set of 10 easy problems to choose from. We will only use one easy problem in each contest, so we can choose our problems in $(10)_3 = \frac{10!}{(10-3)!}$ different ways.

Permutation (variation) with repetition

The number of possible choices of *k* objects from a set of *n* objects when order is important and one object can be chosen more than once:

$$n^k$$

For example, if we have 10 different prizes that need to be divided among 5 people, we can do so in 5^{10} ways.

Permutation with repetition

The number of different permutations of *n* objects, where there are n_1 indistinguishable objects of type 1, n_2 indistinguishable objects of type 2,..., and n_k indistinguishable objects of type *k* ($n_1+n_2+\ldots+n_k=n$), is:

$$\binom{n}{n_1, n_2, \ldots, n_k} = C_n(n_1, n_2, \ldots, n_k) = \frac{n!}{n_1! \cdot n_2! \cdot \ldots \cdot n_k!}$$

For example, if we have 97 coders and want to assign them to 5 rooms (rooms 1-4 have 20 coders in each, while the 5th room has 17), then there are $\binom{97}{20, 20, 20, 20, 17} = \frac{97!}{20! \cdot 20! \cdot 20! \cdot 20! \cdot 17!}$ possible ways to do it.

Combinations without repetition

In combinations we choose a set of elements (rather than an arrangement, as in permutations) so the order doesn't matter. The number of different k-element subsets (when each element can be chosen only once) of n-element set is:

$$\binom{n}{k} = C_k^n = \frac{n!}{k! \cdot (n-k)!}$$

For example, if we have 7 different colored balls, we can choose any 3 of them in $\binom{7}{3} = \frac{7!}{3! \cdot (7-3)!}$ different ways.

Combination with repetition

Let's say we choose k elements from an n-element set, the order doesn't matter and each element can be chosen more than once. In that case, the number of different combinations is:

$$f_k^n = \binom{n+k-1}{k} = \frac{(n+k-1)!}{k! \cdot (n-1)!}$$

For example, let's say we have 11 identical balls and 3 different pockets, and we need to calculate the number of different divisions of these balls to the pockets. There would be $f_{11}^3 = \binom{3+11-1}{11} = \frac{(3+11-1)!}{11! \cdot (3-1)!}$ different combinations.

It is useful to know that f_k^n is also the number of integer solutions to this equation:

$$x_1 + x_2 + \ldots + x_n = k, x_i \geq 0 (i \in \overline{1,n})$$

Why? It's easy to prove. Consider a vector (1, 1, …, 1) consisting of (n+k-1) ones, in which we want to substitute n-1 ones for zeroes in such way that we'll get n groups of ones (some of which may be empty) and the number of ones in the i^{th} group will be the value of x_i:

$$(\underbrace{1,\ldots,1}_{x_1},0,\underbrace{1,\ldots,1}_{x_2},0,\ldots,0,\underbrace{1,\ldots,1}_{x_n})$$

The sum of x_i will be k, because k ones are left after substitution.

The Basics

Binary vectors

Some problems, and contest problems are no exception, can be reformulated in terms of binary vectors. Accordingly, some knowledge of the basic combinatorial properties of binary vectors is rather important. Let's have a look at some simple things associated with them:

1. Number of binary vectors of length n: 2^n.

2. Number of binary vectors of length n and with k '1' is $\binom{n}{k}$.

We just choose k positions for our '1's.

3. The number of ordered pairs (a, b) of binary vectors, such that the distance between them (k) can be calculated as follows: $\binom{n}{k} \cdot 2^n$.
The distance between a and b is the number of components that differs in a and b -- for example, the distance between (0, 0, 1, 0) and (1, 0, 1, 1) is 2).

Let a = (a_1, a_2, …a_n), b = (b_1, b_2, …b_n) and distance between them is k. Next, let's look at the sequence of pairs (a_1, b_1), (a_2, b_2), … (a_n, b_n). There are exactly k indices i in which $a_i \neq b_i$. They can be (0,1) or (1,0), so there are 2 variants, and n-k can be either (0,0) or (1,1), for another 2

variants. To calculate the answer we can choose k indices in which vectors differs in $\binom{n}{k}$ ways, then we choose components that differs in 2^k ways and components that are equal in 2^{n-k} ways (all of which use the permutation with repetition formula), and in the end we just multiply all these numbers and get $\binom{n}{k} \cdot 2^k \cdot 2^{n-k} = \binom{n}{k} \cdot 2^n$.

Delving deeper

Now let's take a look at a very interesting and useful formula called the inclusion-exclusion principle (also known as the sieve principle):

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{i=1}^{n} (-1)^{i-1} \sum_{\substack{I \subseteq 1,n, \\ |I|=i}} \left|\bigcap_{j \in I} A_j\right|$$

This formula is a generalization of:

$$|A \cup B| = |A| + |B| - |A \cap B|$$

There are many different problems that can be solved using the sieve principle, so let's focus our attention on one of them. This problem is best known as "Derangements". A derangement of the finite set X is a bijection from X into X that doesn't have fixed points. A small example: For set X = {1, 2, 3} bijection {(1,1), (2,3), (3,2)} is not derangement, because of (1,1), but bijection {(1,2), (2,3), (3,1)} is derangement. So let's turn back to the problem, the goal of which is to find the number of derangements of n-element set.

We have X = {1, 2, 3,…, n}. Let:

- A be the set of all bijections from X into X, $|A|=n!$,
- A_0 be the set of all derangements of X,
- A_i ($i \in X$) be the set of bijections from X into X that have (i,i),
- A_I ($I \subseteq X$) be the set of bijections from X into X that have (i,i) $\forall i \subseteq I$, so $A_I = \bigcap_{i \in I} A_i$ and $|A_I|=(n-|A_I|)!$.

In formula we have sums $\sum_{\substack{I \subseteq 1,n, \\ |I|=i}} \left|\bigcap_{j \in I} A_j\right|$, in our case we'll have $\sum_{\substack{I \subseteq 1,n, \\ |I|=i}} \left|\bigcap_{j \in I} A_j\right| = \sum_{\substack{I \subseteq X, \\ |I|=i}} |A_I|$, so let's calculate them:

$$\sum_{\substack{I \subseteq X, \\ |I|=i}} |A_I| = \binom{n}{i} (n-i)! = \frac{n!}{i!}$$

(because there are exactly $\binom{n}{i}$ i-element subsets of X).

Now we just put that result into the sieve principle's formula:

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{i=1}^{n} (-1)^{i-1} \sum_{\substack{I \subseteq X, \\ |I|=i}} |A_I| = \sum_{i=1}^{n} (-1)^{i-1} \frac{n!}{i!} = n! \sum_{i=1}^{n} \frac{(-1)^{i-1}}{i!}$$

And now the last step, from $A_0 = A \setminus \bigcup_{i=1}^{n} A_i$ we'll have the answer:

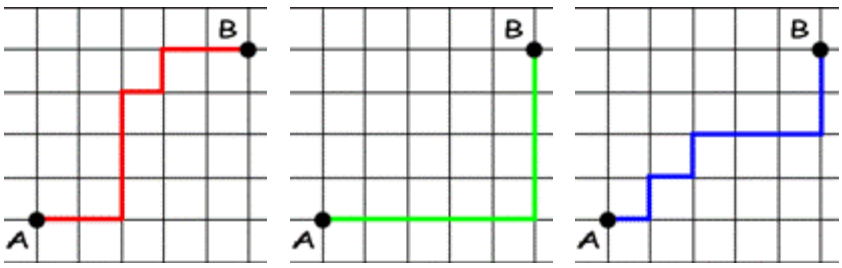
$$|A_0| = |A| - \left|\bigcup_{i=1}^{n} A_i\right| = n! - n! \sum_{i=1}^{n} \frac{(-1)^{i-1}}{i!} = n! \sum_{i=0}^{n} \frac{(-1)^i}{i!}.$$

And the last remark:

$$n!\sum_{i=0}^{n}\frac{(-1)^i}{i!}=\left[\frac{n!}{e}\right].$$

This problem may not look very practical for use in TopCoder problems, but the thinking behind it is rather important, and these ideas can be widely applied.

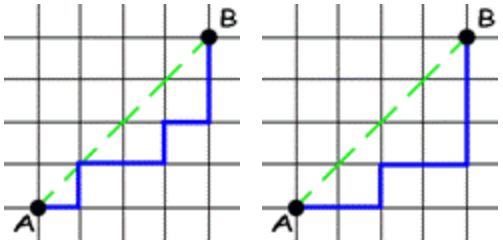
Another interesting method in combinatorics -- and one of my favorites, because of its elegance -- is called method of paths (or trajectories). The main idea is to find a geometrical interpretation for the problem in which we should calculate the number of paths of a special type. More precisely, if we have two points A, B on a plane with integer coordinates, then we will operate only with the shortest paths between A and B that pass only through the lines of the integer grid and that can be done only in horizontal or vertical movements with length equal to 1. For example:



All paths between A and B have the same length equal to n+m (where n is the difference between x-coordinates and m is the difference between y-coordinates). We can easily calculate the number of all the paths between A and B as follows:

$$\binom{n+m}{m} \text{ or } \binom{n+m}{n}.$$

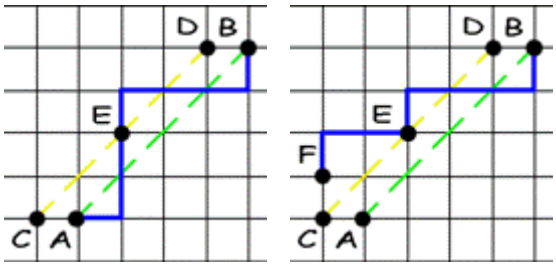
Let's solve a famous problem using this method. The goal is to find the number of Dyck words with a length of 2n. What is a Dyck word? It's a string consisting only of n X's and n Y's, and matching this criteria: each prefix of this string has more X's than Y's. For example, "XXYY" and "XYXY" are Dyck words, but "XYYX" and "YYXX" are not.



Let's start the calculation process. We are going to build a geometrical analog of this problem, so let's consider paths that go from point A(0, 0) to point B(n, n) and do not cross segment AB, but can touch it (see examples for n=4).

It is obvious that these two problems are equivalent; we can just build a bijection in such a way: step right - 'X', step up - 'Y'.

Here's the main idea of the solution: Find the number of paths from A to B that cross segment AB, and call them "incorrect". If path is "incorrect" it has points on the segment CD, where C = (0, 1), D = (n-1, n). Let E be the point nearest to A that belongs to CD (and to the path). Let's symmetrize AE, part of our "incorrect" path with respect to the line CD. After this operation we'll get a path from F = (-1, 1) to B.



It should be easy to see that, for each path from F to B, we can build only one "incorrect" path from A to B, so we've got a bijection. Thus, the number of "incorrect" paths from A to B is $\binom{2n}{n-1}$. Now we can easily get the answer, by subtracting the number of "incorrect" paths from all paths:

$$\binom{2n}{n}-\binom{2n}{n-1}=\frac{(2n)!}{n!\cdot n!}-\frac{(2n)!}{(n-1)!\cdot(n+1)!}=\frac{\binom{2n}{n}}{n+1}$$

This number is also known as n's Catalan number: C_n is the number of Dyck words with length 2n. These numbers also appear in many other problems, for example, C_n counts the number of correct arrangements of n pairs of parentheses in the expression, and C_n is also the number of the possible triangulations of a polygon with (n+2) vertices, and so on.

Using recurrence relations

Recurrence relations probably deserves their own separate article, but I should mention that they play a great role in combinatorics. Particularly with regard to TopCoder, most calculation problems seem to require coders to use some recurrence relation and find the solution for the values of parameters.

If you'd like to learn more, check out these tutorials: An Introduction to Recursion, Recursion, Part 2, and Dynamic Programming: From novice to advanced. Done reading? Let's take a look at some examples.

ChristmasTree (SRM 331 Division Two – Level Three):

We'll use DP to solve this -- it may not be the best way to tackle this problem, but it's the easiest to understand. Let **cnt[lev][r][g][b]** be the number of possible ways to decorate the first **lev** levels of tree using **r** red, **g** green and **b** blue baubles. To make a recurrent step calculating **cnt[lev][r][g][b]** we consider 3 variants:

cnt[lev][r][g][b]=

1) we fill the last level with one color (red, green or blue), so just:

= cnt [lev-1][r-lev][g][b]+ cnt[lev-1][r][g-lev][b]+ cnt[lev-1][r][g][b-lev]+ ;

2) if (lev%2 == 0) we fill the last level with two colors (red+green, green+blue or red+blue), then we calculate the number of possible decorations using the *Permutation with repetition* formula. We'll get $\frac{lev!}{(lev/2)!(lev/2)!}$ possible variants for each two colors, so just

$\frac{lev!}{(lev/2)!(lev/2)!}$(cnt[lev-1][r-lev/2][g-lev/2][b]+...+cnt[lev-1][r][g-lev/2][b-lev/2])+;

3) if (lev%3 == 0) we fill the last level with three colors and, again using the *Permutation with repetition* formula, we'll get $\frac{lev!}{(lev/3)!(lev/3)!(lev/3)!}$ possible variants, so we'll get:

$$+\frac{lev!}{(lev/3)!(lev/3)!(lev/3)!}\cdot \text{cnt[lev-1][r-lev/3][g-lev/3][b-lev/3]}$$

(all cnt[l][i][j][k] with negative indices are 0).

DiceGames (SRM 349 Division One – Level Two):

First we should do some preparation for the main part of the solution, by sorting **sides** array in increasing order and calculating only the formations where the numbers on the dice go in non-decreasing order. This preparation saves us from calculating the same formations several times (see SRM 349 - Problem Set & Analysis for additional explanation). Now we will only need to build the recurrence relation, since the implementation is rather straightforward. Let **ret[i][j]** be the number of different formations of the first *i* dice, with the last dice equal to *j* (so $i \in \overline{0, n-1}, j \in \overline{0, sides[i]-1}$, where n is the number of elements in **sides**). Now we can simply write the recurrence relation:

$$ret[i][j] = \sum_{k=0}^{j} ret[i-1][k], ret[0][i] = 1 (i \in \overline{0, sides[0]-1})$$

The answer will be $\sum_{i=0}^{sides[n-1]-1} ret[n-1][i]$.

Conclusion

As this article was written for novices in combinatorics, it focused mainly on the basic aspects and methods of counting. To learn more, I recommend you check out the reference works listed below, and keep practicing – both in TopCoder SRMs and pure mathematical problems. Good luck!

Introduction

This article presents a new approach for computing maximum flow in a graph. Previous articles had concentrated on finding maximum flow by finding augmenting paths. **Ford-Fulkerson** and **Edmonds-Karp** algorithms belong to that class. The approach presented in this article is called **push-relabel**, which is a separate class of algorithms. We'll look at an algorithm first described by **Andrew V. Goldberg** and **Robert E. Tarjan**, which is not very hard to code and, for dense graphs, is much faster than the augmenting path algorithms. If you haven't yet done so, I'd advise you to review the articles on graph theory and maximum flow using augmenting paths for a better understanding of the basic concepts on the two topics.

The Standard Maximum Flow Problem

Let **G** = **(V,E)** be a directed graph with vertex set **V** and edge set **E**. Size of set **V** is **n** and size of set **E** is **m**. **G** has two distinguished vertices, a source **s** and a sink **t**. Each edge **(u,v) ε E** has a capacity **c(u,v)**. For all edges **(u,v) ∉E**, we define **c(u,v) = 0**. A flow f on **G** is a real valued function satisfying the following constraints:

1. **Capacity: f(v,w) ≤ c(v,w) ∨ (v,w) ∈ V × V**
2. **Anti-symmetry: f(v,w) = – f(w,v) ∨ (v,w) ∈ V × V**
3. **Flow Conservation: ∑ u ∈ V f(u,v) = 0 ∨ v ∈ V - {s,t}**

The value of a flow f is the net flow into the sink i.e. **| f | = ∑ u ∈ V f(u,t)** . **Figure 1** below shows a flow network with the edges marked with their capacities. Using this network we will illustrate the steps in the algorithm.

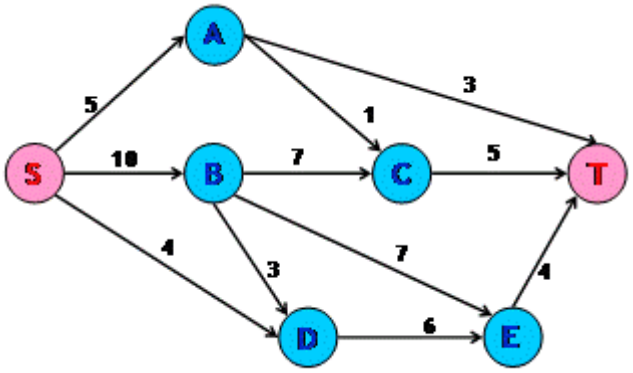


Figure 1 : Flow Network with Capacities

Intuitive Approach Towards the Algorithm

Assume that we have a network of water tanks connected with pipes in which we want to find the maximum flow of water from the source tank to the sink tank. Each of these water tanks are arbitrarily large and will be used for accumulating water. A tank is said to be overflowing if it has water in it. Tanks are at a height from the ground level. The edges can be assumed to be pipes connecting these water tanks. The natural action of water is to flow from a higher level to a lower level. The same holds for this algorithm. The height of the water tank determines the direction in which water will flow. We can push **new flow** from a tank to another tank that is downhill from the first tank, i.e. to tanks that are at a lesser height than the first tank. We need to note one thing here, however:**The flow from a lower tank to a higher tank might be positive.** Present height of a tank only determines the direction of new flow.

We fix the initial height of the source **s** at **n** and that of sink **t** at **0**. All other tanks have initial height **0**, which increases with time. Now send as much as possible flow from the source toward the sink. Each outgoing pipe from the source s is filled to capacity. We will now examine the tanks other than **s** and **t**. The flow from overflowing tanks is pushed downhill. If an overflowing tank is at the same level or below the tanks to which it can push flow, then this tank is raised just enough so that it can push more flow. If the sink t is not reachable from an overflowing tank, we then send this excess water back to the source. This is done by raising the overflowing tank the fixed height **n** of the source. Eventually all the tanks except the source **s** and the sink **t** stop overflowing. At this point the flow from the source to the sink is actually the max-flow.

Mathematical Functions

In this section we'll examine the mathematical notations required for better understanding of the algorithm.

1. **Preflow** - Intuitively preflow function gives the amount of water flowing through a pipe. It is similar to the flow function. Like flow, it is a function **f: V × V → R**. It also follows the capacity and anti-symmetry constraints. But for the preflow function, the conservation constraint is weakened.

∑ u ∈ V f(u,v) ≥ 0 ∨ v ∈ V - {s,t}

That is the total net flow at a vertex other than the source that is non-negative. During the course of the algorithm, we will manipulate this function to achieve the maximum flow.

2. **Excess Flow** - We define the excess flow **e** as **e(u) = f(V,u)**, the net flow into **u**. A vertex **u ∈ V-{s,t}** is overflowing / active if **e(u) > 0**.
3. **Residual Capacity** - We define residual capacity as function **cf: V × V → R** where

$$cf(u,v) = c(u,v) - f(u,v)$$

If **cf(u,v) > 0**, then **(u,v)** is called a **residual edge**. Readers would have come across this concept in augmenting path algorithms as well.

4. **Height** - This function is defined as **h: V → N**. It denotes the height of a water tank. It has the following properties -
   - **h(s) = n**
   - **h(t) = 0**
   - **h(u) ≤ h(v) + 1 for every residual edge (u,v).**

We will prove the last property while discussing the correctness of the algorithm.

Basic Operations

Following are the three basic functions that constitute the algorithm:

1. **Initialization** – This is carried out once during the beginning of the algorithm. The height for all the vertices is set to zero except the source for which the height is kept at **n**. The preflow and the excess flow functions for all vertices are set to zero. Then all the edges coming out the source are filled to capacity. **Figure 2** shows the network after initialization.

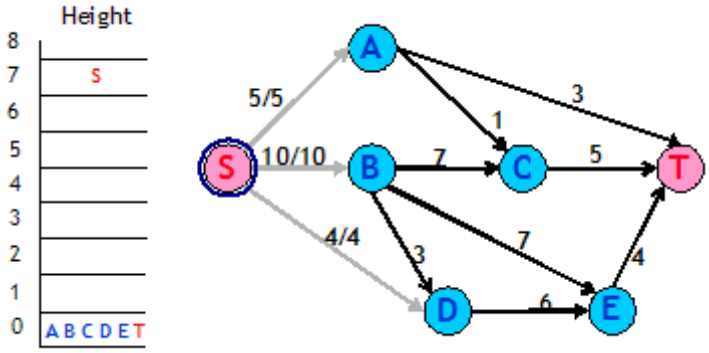


Figure 2 : Network after Initialization

```
void initialize_preflow() {
  memset(h,0,sizeof(int)*n);
  h[s] = n;
  memset(e,0,sizeof(int)*n);
  for(int i=n-1;i>=0;--i)
    memset(f[i],0,sizeof(int)*n);
  for(int i=0;i<G[s].size();++i) {
    int v = G[s][i];
    f[s][v] = c[s][v];
    f[v][s] =-c[s][v];
    e[v] = c[s][v];
    e[s] -= c[s][v];
    cf[s][v] = c[s][v] - f[s][v];
    cf[v][s] = c[v][s] - f[v][s];
  }
}
```

Figure 3 : Code for Initialization

2. **push(u,v)** - This operation pushes flow from an overflowing tank to a tank to which it has a pipe that can take in more flow and the second tank is at a lower height than the first one. In other words, if vertex **u** has a positive excess flow, **cf(u,v) > 0** and **h(u) > h(v)**, then flow can be pushed onto the residual edge **(u,v)**. The amount of the flow pushed is given by **min(e(u),cf(u,v))**.

   Figure 4 shows a vertex **B** that has an excess flow of **10**. It makes two pushes. In the first push, **7** units of flow are pushed to **C**. In the second push, **3** units of flow are pushed to **E. Figure 5** illustrates the final result.

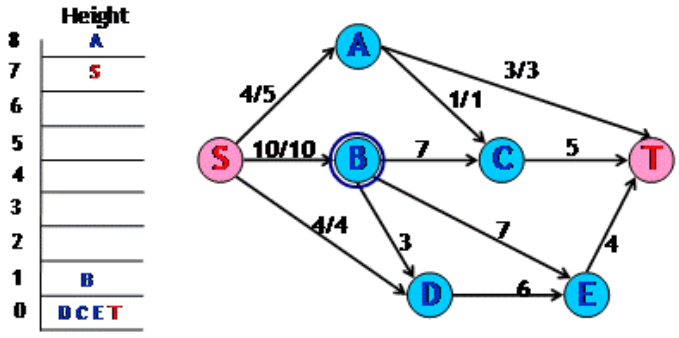


Figure 4 : Network before pushing flow from B

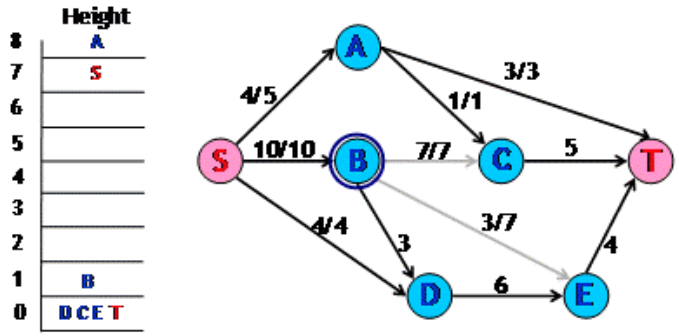


Figure 5 : Network pushing flow from B

```
void push(int u,int v) {
  temp = min(e(u),cf[u][v]);
  f[u][v] = f[u][v] + temp;
  f[v][u] = -f[u][v];
  e[u] = e[u] - temp;
  e[v] = e[v] + temp;
  cf[u][v] = c[u][v] - f[u][v];
  cf[v][u] = c[v][u] - f[v][u];
}
```

Figure 6 : Code for Push sub-routine

3. **relabel(u)** - This operation raises the height of an overflowing tank that has no other tanks downhill from it. It applies if **u** is overflowing and **h(u) ≤ h(v)** ∀ residual edges **(u,v)** i.e. on all the residual edges from **u**, flow cannot be pushed. The height of the vertex **u** is increased by **1** more than the minimum height of its neighbor to which **u** has a residual edge.

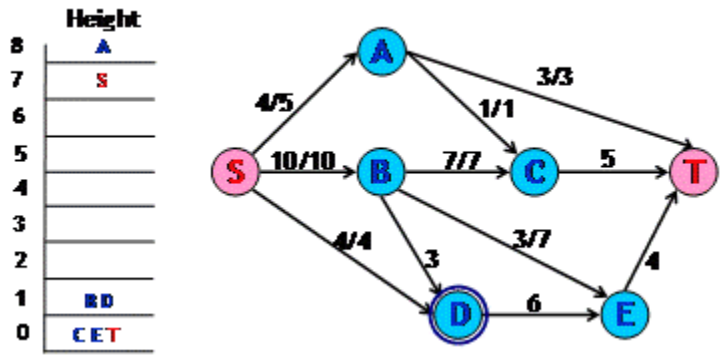


Figure 7 : Network after relabeling D

In **Figure 4**, pick up vertex **D** for applying the push operation. We find that it has no vertex that is downhill to it and the edge from **D** to that vertex has excess capacity. So we relabel **D** as shown in **Figure 5.**

```
1  void relabel(int u){
2    int temp = -1;
3    for(int i=0;i<G[u].size();++i) {
4      v = G[u][i];
5      if(cf[u][v] > 0) {
6        if(temp == -1 || temp > h[v])
7          temp = h[v];
8      }
9    }
10   h[u] = 1 + temp;
11 }
```

Figure 8 : Code for Relabel sub-routine

Generic Algorithm
The generic algorithm is as follows:

```
void maxflow() {
    initialize_preflow();
    while(there is an operation that can be carried out)
      Select an operation and perform it.
}
```

The value **e(t)** will represent the maximum flow. We now try to see why this algorithm would work. This would need two observations that hold at all times during and after the execution of the algorithm.

1. **A residual edge from u to v implies h(u) ≤ h(v) + 1** -We had earlier introduced this as a property of the height function. Now we make use of induction for proving this property.
   - Initially residual edges are from vertices of height 0 to the source that is at height **n**.
   - Consider a vertex **u** getting relabeled and **v** is the vertex of minimum height to which **u** has a residual edge. After relabeling, the property holds for the residual edge **(u,v)**. For any other residual edge **(u,w)**, **h(v) ≤h(w)**. So after relabeling the property will hold for residual edge **(u,w)**. For a residual edge, **(w,u)**, since **u**'s height only goes up, therefore the property will continue to hold trivially.
   - Consider a push operation from **u** to **v**. After the operation, edge (v,u) will be present in the residual graph. Now **h(u) > h(v) ∧ h(u) ≤ h(v) + 1**

     → h(u) = h(v) + 1
     → h(v) = h(u) – 1
     → h(v) _ h(u) + 1

2. **There is no path for source to sink in the residual graph** - Let there be a simple path **{v_0 , v_1 , . . . , v_{k-1}, v_k }** from **v_0 = s** to **v_k = t**. Then, as per the above observation,

   h(v i) ≤ h(v i+1) + 1
   → h(s) ≤ h(t) + k
   → n ≤ k
   → n < k + 1

   This violates the simple path condition as the path has **k+1** vertices. Hence there is no path from source to sink.

When the algorithm ends, no vertex is overflowing except the source and the sink. There is also no path from source to sink. This is the same situation in which an augmenting path algorithm ends. Hence, we have maximum flow at the end of the algorithm.

Analysis
The analysis requires proving certain properties relating to the algorithm.

$$\begin{aligned}\sum_{v\in S} e(v) &= \sum_{w\in V, v\in S} f(w,v) \\ &= \sum_{w\in V-S, v\in S} f(w,v) + \sum_{w\in S, v\in S} f(w,v) \\ &= \sum_{w\in V-S, v\in S} f(w,v) \\ &\leq 0\end{aligned}$$

1. **s is reachable from all the overflowing vertices in the residual graph** - Consider **u** as an overflowing and **S** as the set of all the vertices that are reachable from **u** in the residual graph. Suppose **s ∉ 2S** . Then for every vertex pair **(v,w)** such that **v ∈ S** and **w ∈ V-S** , **f(w,v) ≤ 0**. Because if **f(w,v) > 0**, then **cf(v,w) > 0** which implies **w** is reachable from **u**. Thus, since **e(v) ≤ 0**, for all **v ∈ S, e(v) = 0**. In particular, **e(u) = 0**, which is a contradiction.
2. **The height of a vertex never decreases** - The height of a vertex changes only during the **relabeling** operation. Suppose we relabel vertex **u**. Then for all vertices **v** such that **(u,v)** is a residual edge, we have **h(v) ≤ h(u)**, which clearly means **h(v)+1>h(u)**. So the height of a vertex never decreases.
3. **The height of a vertex can be at maximum 2n-1** - This holds for **s** and **t** since their heights never change. Consider a vertex **u** such that **e(u) > 0**. Then there exists a simple path from **u** to **s** in the residual graph. Let this path be **u = v_0, v_1 , . . . , v_{k-1}, v_k = s**. Then **k** can be at most **n-1**. Now, as per the definition of **h**, **h(v_i) ≤ h(v_{i+1}) + 1**. This would yield **h(u) ≤ h(s) + n – 1 = 2n – 1**.

Now we are in a position to count the number of operations that are carried out during the execution of the code.

- **Relabeling operations** - The height for each vertex other than **s** and **t** can change from **0** to **2n-1**. It can only increase, so there can be utmost **2n-1** relabelings for each vertex. In total there can be a maximum of **(2n-1)(n-2)** relabelings. Each relabeling requires at most **degree(vertex)** operations. Summing this up over all the vertices and over all the relabelings, the total time spent in relabeling operations is **O(nm)**.
- **Saturated Pushes** - A saturating push from **u** to **v**, results in **f(u,v) = c(u,v)**. In order to push flow from **u** to **v** again, flow must be pushed from **v** to **u** first. Since **h(u) = h(v) + 1**, so **v**'s height must increase by at least **2**. Similarly **h(u)** should increase by at least **2** for the next push. Combining this with the earlier result on the maximum height of avertex, the total number of saturating pushes is at most **2n-1** per edge. So that total overall the edges is at most **(2n-1)m < 2nm**.
- **Non-saturated Pushes** - Let **ϕ = ∑ u ∈ V, u is active h(u)**. Each non-saturating push from a vertex **u** to any other vertex **v** causes **ϕ** to decrease by at least **1** since **h(u) > h(v)**. Each saturating push can increase **ϕ** by at most **2n-1** since it could be that **v** was not active before. The total increase in **ϕ** due to saturating pushes is at most **(2n-1)(2nm)**. The total increase in **ϕ** due to relabeling operation is at most **(2n-1)(n-2)**. Therefore, the total number of non-saturating pushes is at most **(2n-1)(2nm) + (2n-1)(n-2)≤ $4n^2m$**.

Thus the generic algorithm takes **O(n^2m)** operations in total. Since each push operation requires **O(1)** time, hence the running time of the algorithm will also be **O(n^2m)** if the condition given in the while loop can be tested in **O(1)** time. The next section provides an implementation for doing the same. In fact, by ordering the operations in a particular manner, a more rigorous analysis proves that a better time bound can be achieved.

First-in First-out Algorithm

We will make use of a first-in, first-out strategy for selecting vertices on which push/relabel operation will be performed. This is done by creating a queue of vertices that are overflowing. Initially all the overflowing vertices are put into the queue in any order. We will run the algorithm till the queue becomes empty.

In every iteration the vertex at the front of the queue is chosen for carrying out the operations. Let this vertex be **u**. Consider all the edges of **u**, both those that are incident on **u** and those that are incident on other vertices from **u**. These are edges along which **u** can potentially push more flow. Go through these edges one by one, pushing flow along edges that have excess capacity. This is done until either **u** becomes inactive or all the edges have been explored. If during the iteration any new vertex starts to overflow, then add that vertex to the end of the queue. If at the end of the iteration **u** is still overflowing, then it means **u** needs a relabeling. Relabel **u** and start the next iteration with **u** at the front of the queue. If any time during the process or at end of it **u** becomes inactive, remove **u** from the queue. This process of pushing excess flow from a vertex until it becomes inactive is called **discharging a vertex**.

```
void maxflow() {
  initialize_preflow();
  queue<int> q;
  char *l = new char[n];
  int u,v,m;
  memset(l,0,sizeof(char)*n);
  for(int i=0;i<G[s].size();++i) {
    if(G[s][i] != t) {
      q.push(G[s][i]);
      l[G[s][i]] = 1;
    }
  }
  while(q.size() != 0) {
    u = q.front();
    m = -1;
    for(int i=0;i<G[u].size() && e[u] > 0;++i) {
      v = G[u][i];
      if(cf[u][v] > 0) {
        if(h[u] > h[v]) {
          push(u,v);
          if(l[v] == 0 && v != s && v != t) {
            l[v] = 1;
            q.push(v);
          }
        }
        else if(m == -1) m = h[v];
        else m = min(m,h[v]);
      }
    }
    if(e[u] != 0) h[u] = 1 + m;
    else {
      l[u] = 0;
      q.pop();
    }
  }
}
```

Figure 9 : Code for First-In First-Out Algorithm

Analysis of First-In First-Out Algorithm

To analyze the strategy presented above, the concept of a **pass over a queue** needs to be defined. **Pass 1** consists of discharging the vertices added to the queue during initialization. **Pass i + 1** consists of discharging vertices added during the **Pass i**.

- **The number of passes over the queue is at most $4n^2$** - Let **ϕ** = **max {h(u) |u is active|}**. If no heights changes during a given pass, each vertex has its excess moved to vertices that are lower in height. Hence **ϕ** decreases during the pass. If **ϕ** does not change, then there is at least one vertex whose height increases by at least 1. If **ϕ** increases during a pass, then some vertex's height must have increased by as much as the increase **ϕ**. Using the proof about the maximum height of the vertex, the maximum number of passes in which **ϕ** increases or remains same is $2n^2$. The total number of passes in which **ϕ** decreases is also utmost $2n^2$.Thus the total number of passes is utmost $4n^2$.
- **The number of non-saturating pushes is at most $4n^3$** - There can be only one non-saturating push per vertex in a single pass. So the total number of non-saturating pushes is at most $4n^3$.

On combining all the assertions, the total time required for running the first-in first-out algorithm is $O(nm + n^3)$ which is $O(n^3)$.

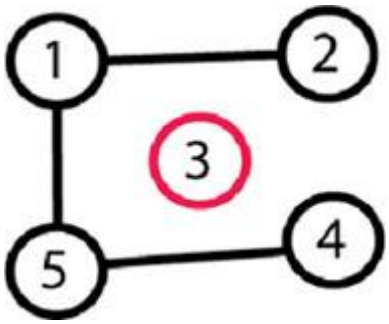
Related Problems

In general any problem that has a solution using max-flow can be solved using **"push-relabel"** approach. Taxi and Quest4 are good problems for practice. More problems can be found in the article on max flow using augmenting paths. In problems where the time limits are very strict, push-relabel is more likely to succeed as compared to augmenting path approach. Moreover, the code size is not very significant. The code provided is of **61 lines** (16 + 36 + 9), and I believe it can be shortened.

In places where only the maximum flow value is required and the actual flow need not be reported, the algorithm can be changed slightly to work faster. The vertices that have heights **≥n** may not be added to the queue. This is because these vertices can never push more flow towards the sink. This may improve the running time by a factor of 2. Note that this will not change the asymptotic order of the algorithm. This change can also be applied in places where the min-cut is required. Let **(S,T)** be the min-cut. Then, **T** contains the vertices that reachable from **t** in the residual graph.

Introduction
Many times the efficiency of an algorithm depends on the data structures used in the algorithm. A wise choice in the structure you use in solving a problem can reduce the time of execution, the time to implement the algorithm and the amount of memory used. During SRM competitions we are limited to a time limit of 2 seconds and 64 MB of memory, so the right data structure can help you remain in competition. While some Data Structures have been covered before, in this article we'll focus on data structures for disjoint sets.

The problem
Let's consider the following problem: In a room are N persons, and we will define two persons are friends if they are directly or indirectly friends. If A is a friend with B, and B is a friend with C, then A is a friend of C too. A group of friends is a group of persons where any two persons in the group are friends. Given the list of persons that are directly friends find the number of groups of friends and the number of persons in each group. For example N = 5 and the list of friends is: 1-2, 5-4, and 5-1. Here is the figure of the graph that represents the groups of friends. 1 and 2 are friends, then 5 and 4 are friends, and then 5 and 1 are friends, but 1 is friend with 2; therefore 5 and 2 are friends, etc.



In the end there are 2 groups of friends: one group is {1, 2, 4, 5}, the other is {3}.

The solution
This problem can be solved using BFS, but let's see how to solve this kind of problem using data structures for disjoint sets. First of all: a disjoint-set data structure is a structure that maintains a collection S1, S2, S3, …, Sn of dynamic disjoint sets. Two sets are disjoint if their intersection is null. For example set {1, 2, 3} and set {1, 5, 6} aren't disjoint because they have in common {1}, but the sets {1, 2, 3} and {5, 6} are disjoint because their intersection is null. In a data structure of disjoint sets every set contains a **representative**, which is one member of the set.

Let's see how things will work with sets for the example of the problem. The groups will be represented by sets, and the representative of each group is the person with the biggest index. At the beginning there are 5 groups (sets): {1}, {2}, {3}, {4}, {5}. Nobody is anybody's friend and everyone is the representative of his or her own group.

The next step is that 1 and 2 become friends, this means the group containing 1 and the group with 2 will become one group. This will give us these groups: {1, 2} , {3}, {4}, {5}, and the representative of the first group will become 2. Next, 5 and 4 become friends. The groups will be {1,2}, {3}, {4, 5}. And in the last step 5 and 1 become friends and the groups will be {1, 2, 4, 5}, {3}. The representative of the first group will be 5 and the representative for second group will be 3. (We will see why we need representatives later). At the end we have 2 sets, the first set with 4 elements and the second with one, and this is the answer for the problem example: 2 groups, 1 group of 4 and one group of one.

Perhaps now you are wondering how you can check if 2 persons are in the same group. This is where the use of the representative elements comes in. Let's say we want to check if 3 and 2 are in the same group, we will know this if the representative of the set that contains 3 is the same as the representative of the set that contains 2. One representative is 5 and the other one is 3; therefore 3 and 2 aren't in same groups of friends.

Some operations
Let's define the following operations:

- CREATE-SET(x) – creates a new set with one element {x}.
- MERGE-SETS(x, y) – merge into one set the set that contains element x and the set that contains element y (x and y are in different sets). The original sets will be destroyed.
- FIND-SET(x) – returns the representative or a pointer to the representative of the set that contains element x.

The solution using these operations
Let's see the solution for our problem using these operations:

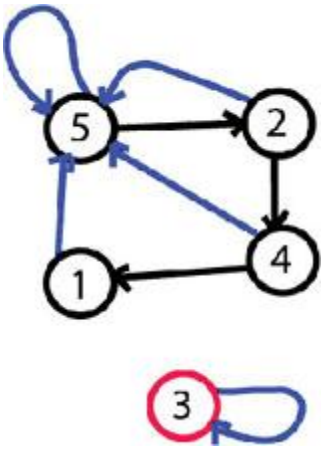
Read N;

for (each person x from 1 to N) CREATE-SET(x)
for (each pair of friends (x y)) if (FIND-SET(x) != FIND-SET(y)) MERGE-SETS(x, y)

Now if we want to see if 2 persons (x, y) are in same group we check if FIND-SET(x) == FIND-SET(y).

We will analyze the running time of the disjoint-set data structure in terms of N and M, where N is the number of times that CREATE-SET(x) is called and M is the total number of times that CREATE-SET(x), MERGE-SETS(x, y) and FIND-SET(x) are called. Since the sets are disjoint, each time MERGE-SETS(x, y) is called one set will be created and two will be destroyed, giving us one less set. If there are n sets after n-1 calls of MERGE-SETS(x,y) there will remain only one set. That's why the number of MERGE-SETS(x,y) calls is less than or equal to the number of CREATE-SET(x) operations.

Implementation with linked lists
One way to implement disjoint set data structures is to represent each set by a linked list. Each element (object) will be in a linked list and will contain a pointer to the next element in the set and another pointer to the representative of the set. Here is a figure of how the example of the problem will look like after all operations are made. The blue arrows are the pointers to the representatives and the black arrows are the pointers to the next element in the sets. Representing sets with linked lists we will obtain a complexity of O(1) for CREATE-SET(x) and FIND-SET(x). CREATE-SET(x) will just create a new linked list whose only element (object) is x, the operation FIND-SET(x) just returns the pointer to the representative of the set that contains element (object) x.



Now let's see how to implement the MERGE-SETS(x, y) operations. The easy way is to append x's list onto the end of y's list. The representative of the new set is the representative of the original set that contained y. We must update the pointer to the representative for each element (object) originally on x's list, which takes linear time in terms of the length of x's list. It's easy to prove that, in the worst case, the complexity of the algorithm will be O(M^2) where M is the number of operations MERGE-SETS(x, y). With this implementation the complexity will average O(N) per operation where N represents the number of elements in all sets.

The "weighted union heuristic"
Let's see how a heuristic will make the algorithm more efficient. The heuristic is called "a weighted-union heuristic." In this case, let's say that the representative of a set contains information about how many objects (elements) are in that set as well. The optimization is to always append the smaller list onto the longer and, in case of ties, append arbitrarily. This will bring the complexity of the algorithm to O(M + NlogN) where M is the number of operations (FIND-SET, MERGE-SETS, CREATE-SETS) and N is the number of operations CREATE-SETS. I will not prove why the complexity is this, but if you are interested you can find the proof in the resources mentioned at the end of the article.

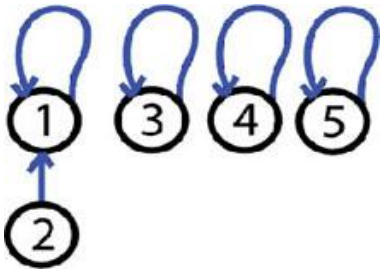
So far we reach an algorithm to solve the problem in O(M + NlogN) where N is the number of persons and M is the number of friendships and a memory of O(N). Still the BFS will solve the problem in O(M + N) and memory of O(N + M). We can see that we have optimized the memory but not the execution time.

Next step: root trees
The next step is to see what we can do for a faster implementation of disjoint set data structures. Let's represent sets by rooted trees, with each node containing one element and each tree representing one set. Each element will point only to its parent and the root of each tree is the representative of that set and its own parent. Let's see, in steps, how the trees will look for the example from the problem above.

Step 1: nobody is anybody friend

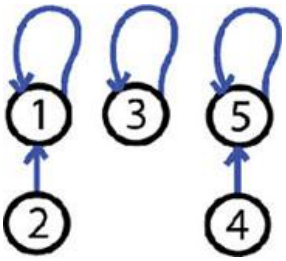
We have 5 trees and each tree has a single element, which is the root and the representative of that tree.

Step 2: 1 and 2 are friends, MERGE-SETS(1, 2):



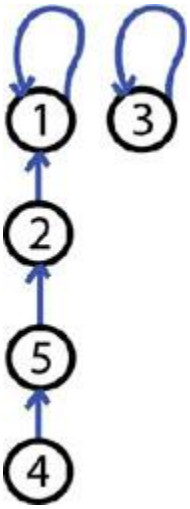
The operation made is MERGE-SETS(1, 2). We have 4 trees one tree contain 2 elements and have the root 1. The other trees have a single element.

Step 3: 5 and 4 are friends, MERGE-SETS(5, 4)



The operation made is MERGE-SETS(5, 4). Now we have 3 trees, 2 trees with 2 elements and one tree with one element.

Step 4: 5 and 1 are friends, MERGE-SETS(5, 1)



The operation made is MERGE-SETS(5, 1). Now we have 2 trees, one tree has 4 elements and the other one has only one element.

As we see so far the algorithm using rooted trees is no faster than the algorithm using linked lists.

Two heuristics

Next we will see how, by using two heuristics, we will achieve the asymptotically fastest disjoint set data structure known so far, which is almost linear in terms of the number of operations made. These two heuristics are called "**union by rank**" and "**path compression.**" The idea in the first heuristic "**union by rank**" is to make the root of the tree with fewer nodes point to the root of the tree with more nodes. For each node, we maintain a **rank** that approximates the logarithm of the sub-tree size and is also an upper bound on the height of the node. When MERGE-SETS(x, y) is called, the root with smaller rank is made to point to the root with larger rank. The idea in the second heuristic "**path compression,**" which is used for operation FIND-SET(x), is to make each node on the find path point directly to the root. This will not change any ranks.

To implement a disjoint set forest with these heuristics, we must keep track of ranks. With each node x, we keep the integer value *rank[x],* which is bigger than or equal to the number of edges in the longest path between node x and a sub-leaf. When CREATE-SET(x) is called the initial rank[x] will be 0. When a MERGE-SETS(x, y) operation is made then the root of higher rank will become the parent of the root of lower rank – or, in case of tie, we arbitrarily choose one of the roots as the parent and increment its rank.

Let's see how the algorithm will look.

Let P[x] = the parent of node x.

```
CREATE-SET(x)
 P[x] = x
 rank[x] = 0
MERGE-SETS(x, y)
 PX = FIND-SET(X)
 PY =FIND-SET(Y)
 If (rank[PX] > rank[PY]) P[PY] = PX
 Else P[PX] = PY
 If (rank[PX] == rank[PY]) rank[PY] = rank[PY] + 1

And the last operation looks like:

FIND-SET(x)
 If (x != P[x]) p[x] = FIND-SET(P[X])
 Return P[X]
```

Now let's see how the heuristics helped the running time. If we use only the first heuristic "**union by rank**" then we will get the same running time we achieved with the weighted union heuristic when we used lists for representation. When we use both "**union by rank**" and "**path compression,**" the worst running time is O(m α(m,n)), where α(m,n) is the very slowly growing inverse of Ackermann's function. In application α(m,n) <= 4 that's why we can say that the running time is linear in terms of m, in practical situations. (For more details on Ackermann's function or complexity see the references below.)

Back to the problem

The problem from the beginning of the article is solvable in O(N + M) and O(N) for memory using disjoint-set data structure. The difference for time execution is not big if the problem is solved with BFS, but we don't need to keep in memory the vertices of the graph. Let's see if the problem was like: In a room are N persons and you had to handle Q queries. A query is of the form "x y 1," meaning that x is friends with y, or "x y 2" meaning that we are asked to output if x and y are in same group of friends at that moment in time. In this case the solution with disjoint-set data structure is the fastest, giving a complexity of O(N + Q).

Practice

Disjoint-set data structures are a helpful tool for use in different algorithms, or even for solving problems in an SRM. They are efficient and use small amount of memory. They are useful in applications like "Computing the shorelines of a terrain," "Classifying a set of atoms into molecules or fragments," "Connected component labeling in image analysis," and others.

To practice what you've learned, try to solve GrafixMask – the Division 1 500 from SRM211. The idea is to keep track of all the blocks and consider each grid point as a node. Next, take all the nodes that aren't blocked and let (x, y) be the coordinate of the left, right, down or up node, and if (x, y) is not blocked then you do the operation MERGE-SETS(node, node2). You should also try to determine how disjoint-set data structures can be used in the solution of RoadReconstruction from SRM 356. Disjoint-set data structures can also be used in TopographicalImage from SRM 210 and PathFinding, from SRM 156.

I hope you find this data structure to be useful. Good luck in the Arena!

Introduction

There are many algorithms and data structures to index and search strings inside a text, some of them are included in the standard libraries, but not all of them; the trie data structure is a good example of one that isn't.

Let **word** be a single string and let **dictionary** be a large set of words. If we have a dictionary, and we need to know if a single word is inside of the dictionary the tries are a data structure that can help us. But you may be asking yourself, "Why use **tries** if **set** <**string**> and hash tables can do the same?" There are two main reasons:

- The tries can insert and find strings in O(*L*) time (where *L* represent the length of a single word). This is much faster than set , but is it a bit faster than a hash table.
- The **set** <**string**> and the hash tables can only find in a dictionary words that match exactly with the single word that we are finding; the trie allow us to find words that have a single character different, a prefix in common, a character missing, etc.

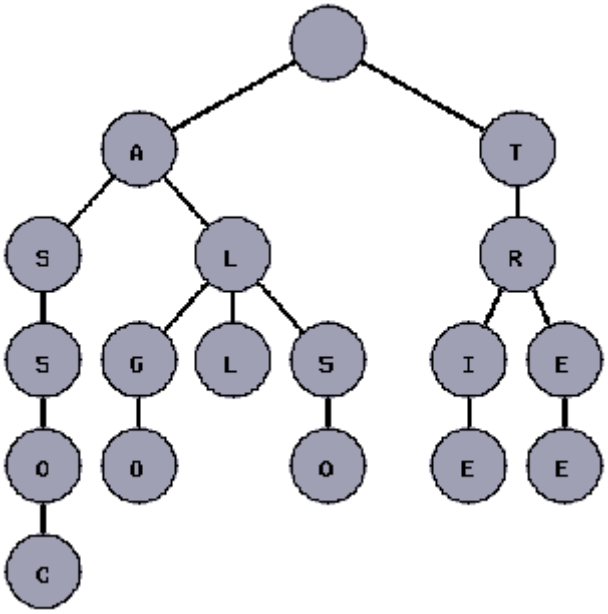
The tries can be useful in TopCoder problems, but also have a great amount of applications in software engineering. For example, consider a web browser. Do you know how the web browser can auto complete your text or show you many possibilities of the text that you could be writing? Yes, with the trie you can do it very fast. Do you know how an orthographic corrector can check that every word that you type is in a dictionary? Again a trie. You can also use a trie for suggested *corrections* of the words that are present in the text but not in the dictionary.

What is a Tree?

You may read about how wonderful the tries are, but maybe you don't know yet what the tries are and why the tries have this name. The word trie is an infix of the word "re**trie**val" because the trie can find a single word in a dictionary with only a prefix of the word. The main idea of the trie data structure consists of the following parts:

- The trie is a tree where each vertex represents a single word or a prefix.
- The root represents an empty string (""), the vertexes that are direct sons of the root represent prefixes of length 1, the vertexes that are 2 edges of distance from the root represent prefixes of length 2, the vertexes that are 3 edges of distance from the root represent prefixes of length 3 and so on. In other words, a vertex that are *k* edges of distance of the root have an associated prefix of length *k*.
- Let **v** and **w** be two vertexes of the trie, and assume that **v** is a direct father of **w**, then **v** must have an associated prefix of **w**.

The next figure shows a trie with the words "tree", "trie", "algo", "assoc", "all", and "also."



Note that every vertex of the tree does not store entire prefixes or entire words. The idea is that the program should remember the word that represents each vertex while lower in the tree.

Coding a Trie

The tries can be implemented in many ways, some of them can be used to find a set of words in the dictionary where every word can be a little different than the target word, and other implementations of the tries can provide us with only words that match exactly with the target word. The implementation of the trie that will be exposed here will consist only of finding words that match exactly and counting the words that have some prefix. This implementation will be pseudo code because different coders can use different programming languages.

We will code these 4 functions:

- addWord. This function will add a single string **word** to the dictionary.
- countPreffixes. This function will count the number of words in the dictionary that have a string **prefix** as prefix.
- countWords. This function will count the number of words in the dictionary that match exactly with a given string **word**.
- Our trie will only support letters of the English alphabet.

We need to also code a structure with some fields that indicate the values stored in each vertex. As we want to know the number of words that match with a given string, every vertex should have a field to indicate that this vertex represents a complete word or only a prefix (for simplicity, a complete word is considered also a prefix) and how many words in the dictionary are represented by that prefix (there can be repeated words in the dictionary). This task can be done with only one integer field words.

Because we want to know the number of words that have like prefix a given string, we need another integer field **prefixes** that indicates how many words have the prefix of the vertex. Also, each vertex must have references to all his possible sons (26 references). Knowing all these details, our structure should have the following members:

```
structure Trie
    integer words;
    integer prefixes;
    reference edges[26];
```

And we also need the following functions:

```
initialize(vertex)
addWord(vertex, word);
integer countPrefixes(vertex, prefix);
integer countWords(vertex, word);
```

First of all, we have to initialize the vertexes with the following function:

```
initialize(vertex)
    vertex.words=0
    vertex.prefixes=0
    for i=0 to 26
        edges[i]=NoEdge
```

The addWord function consists of two parameters, the vertex of the insertion and the word that will be added. The idea is that when a string **word** is added to a vertex **vertex**, we will add **word** to the corresponding branch of **vertex** cutting the leftmost character of word. If the needed branch does not exist, we will have to create it. All the TopCoder languages can simulate the process of cutting a character in constant time instead of creating a copy of the original string or moving the other characters.

```
addWord(vertex, word)
    if isEmpty(word)
        vertex.words=vertex.words+1
    else
        vertex.prefixes=vertex.prefixes+1
        k=firstCharacter(word)
        if(notExists(edges[k]))
            edges[k]=createEdge()
            initialize(edges[k])
        cutLeftmostCharacter(word)
        addWord(edges[k], word)
```

The functions countWords and countPrefixes are very similar. If we are finding an empty string we only have to return the number of words/prefixes associated with the vertex. If we are finding a non-empty string, we should to find in the corresponding branch of the tree, but if the branch does not exist, we have to return 0.

```
countWords(vertex, word)
    k=firstCharacter(word)
    if isEmpty(word)
        return vertex.words
    else if notExists(edges[k])
        return 0
    else
        cutLeftmostCharacter(word)
        return countWords(edges[k], word);

countPrefixes(vertex, prefix)
    k=firstCharacter(prefix)
    if isEmpty(word)
        return vertex.prefixes
    else if notExists(edges[k])
        return 0
    else
        cutLeftmostCharacter(prefix)
        return countWords(edges[k], prefix)
```

Complexity Analysis

In the introduction you may read that the complexity of finding and inserting a trie is linear, but we have not done the analysis yet. In the insertion and finding notice that lowering a single level in the tree is done in constant time, and every time that the program lowers a single level in the tree, a single character is cut from the string; we can conclude that every function lowers *L* levels on the tree and every time that the function lowers a level on the tree, it is done in constant time, then the insertion and finding of a word in a trie can be done in O(*L*) time. The memory used in the tries depends on the methods to store the edges and how many words have prefixes in common.

Other Kinds of Tries

We used the tries to store words with lowercase letters, but the tries can be used to store many other things. We can use bits or bytes instead of lowercase letters and every data type can be stored in the tree, not only strings. Let flow your imagination using tries! For example, suppose that you want to find a word in a dictionary but a single letter was deleted from the word. You can modify the countWords function:

```
countWords(vertex, word, missingLetters)
    k=firstCharacter(word)
    if isEmpty(word)
        return vertex.word
    else if notExists(edges[k]) and missingLetters=0
        return 0
    else if notExists(edges[k])
        cutLeftmostCharacter(word)
        return countWords(vertex, word, missingLetters-1)
        Here we cut a character but we don't go lower in the tree
    else
        We are adding the two possibilities: the first
        character has been deleted plus the first character is present
        r=countWords(vertex, word, missingLetters-1)
        cutLeftmostCharacter(word)
        r=r+countWords(edges[k], word, missingLetters)
        return r
```

The complexity of this function may be larger than the original, but it is faster than checking all the subsets of characters of a word.

In this article, we'll explore the exciting topic of multidimensional databases, which we'll refer to as MDDBs from now on. MDDBs are a very popular technology in the business intelligence arena, and they allow a company to perform in-depth, strategic analysis on a variety of factors affecting their company. In addition, MDDBs allow analysts in the company to leverage tools with which they're already familiar, such as Microsoft® Excel, to work with and analyze data from MDDBs in a "slice and dice" fashion. The ability to slice and dice the data puts tremendous power in the hands of the business user, and that will become more apparent as we delve into more of the details of MDDBs.

The format of this article will be primarily focused on MDDB concepts over specific MDDB vendors; however we'll look at Hyperion® Essbase in several parts of the article to give you some concrete examples of MDDB technology in practice. In addition, we'll look at some real world examples of MDDBs so that you can get a feel of how they are used in the business arena and the impact they can make.

Having knowledge of relational databases will be helpful to someone reading this article, but it's not required. However, we'll assume that you at least know what a relational database is, because we'll be comparing relational and multidimensional databases. If you need an introduction to relational databases, you can refer to the article here.

Introduction

First, just what in the world are we talking about when we say "multidimensional database"? Well, before we get to that, there are a few terms we need to understand. The first term is online **transaction** processing, commonly referred to as OLTP. As the name implies, **OLTP** consists of transaction-based systems, which frequently use a relational database as the back end data store. OLTP systems are usually focused on quick response times and servicing needs immediately. A good example of an OLTP system is the automatic teller machine (ATM) at your local bank. When you go to make a withdrawal from the ATM, you aren't really interested in analyzing a bunch of data, or at least the people in line behind you hope you aren't! In other words, you are there to make a withdrawal, deposit, etc. and move along so that the next person waiting in line can complete their transaction. The OLTP system needs to expedite that process. This leads us to another term we need to define, **OLAP**, which is stands for online **analytical** processing. For you history buffs, the term OLAP was coined by E.F. Codd & Associates in a white paper in 1994[1]. The OLAP approach is focused on the analysis of data, and typically that data originates from OLTP systems. That's not to say that OLAP technology is "slower" than OLTP technology. But the two are focused on entirely different things. OLTP is focused on "getting the job done", and OLAP is focused on "analyzing the results of the job".

So it's important to understand that there's quite a paradigm shift when we go from relational databases to MDDBs. To give you another more concrete example, think of the order processing system running at a sales warehouse. All day long that system is storing orders, shipping information, inventory, and a lot of other information required for running a sales operation. Moreover, there are many people at the warehouse who are interacting with that OLTP system, recording orders, shipping product, managing late orders, etc. These individuals are concerned with making sure that the operation is running smoothly and that customer needs are met, so it's critical that the OLTP system respond as quickly as possible.

But a sales analyst might ask the question, "How are we doing on our sales this year vs. our budget (i.e. our sales plan)?" The people in the sales warehouse probably aren't going to be able to answer that question. However, it's their OLTP system that holds the data which needs to be analyzed to come up with the answer. The OLTP system has all the orders for all product sold for the year, which is exactly what the sales analyst needs so they can compare it to the budget (probably stored in another system). But there might be billions of records in the OLTP system, so trying to run reports off of the OLTP data is bad for a number of reasons. First and foremost, OLTP systems are usually mission-critical to the business. When these systems slow down or experience failure, the entire business suffers. Have you ever tried to order something off a website and gotten some kind of message like "Our site is currency experiencing problems, please try again later"? Few things are more frustrating. So it's very important that OLTP systems stay online and function at top performance levels, and we want to avoid putting any unnecessary load on those systems.

The sales analyst still needs an answer, so what should we do? Well, that's where OLAP technology comes in. We can take the data from the OLTP system and load it to our OLAP system. Note that when we say "OLAP system", we're really referring to the MDDB. The two terms mean essentially the same thing. Another common term you'll hear used for an MDDB is a "cube", or an "OLAP cube". Really, the terms MDDB, cube, and OLAP cube are used pretty interchangeably, but we will discuss the cube concept shortly. Let's get back to our sales analyst's issue. We would typically perform the load of OLTP data to the MDDB in the early morning hours when no users are using the OLTP system, so as to minimize any kind of potential business impact. In systems that are available 24 hours a day 7 days a week, we should load the OLTP data during non-peak times. The load process is usually a lot more involved and may require "massaging" the data to get it into an analytical form so it can be put into the MDDB. The term for this load process is called ETL (extract/transform/load), and there are many ETL tools out on the market today. But ETL is really beyond the scope of our discussion here. So once the nightly ETL load runs, the OLAP database is loaded and the sales analyst can come in the next morning and find the answer to her question. The beauty of it is that she can do that **without** impacting the OLTP system one bit. The folks in the sales warehouse can continue taking orders and running the business, and the sales analyst can do critical analysis of important data which can help determine how the business is performing.

One thing to keep in mind when discussing OLAP databases is that there is usually data latency in an OLAP cube. In other words, the data is usually from some point in time, like maybe the snapshot of data from the night before. But from an analytical standpoint, this data latency is usually just fine because users are doing analysis like comparing sales trends, etc. and it's not really critical to have up-to-the-minute data from the OLTP system. Note that some of the newer OLAP technology offered by Microsoft Analysis Services allows for nearly real time analysis of business intelligence data[3]. In other words, they have mechanisms where you can keep the OLAP data in sync with the relational data.

So to wrap up this section, we've looked at OLTP and OLAP and the differences between the two. Make sure you have a good grasp of these concepts before moving to the next section, where we'll begin to explore the "cube" concept.

The "Cube"

In the last section, we introduced the term "OLAP cube". The reason the term "cube" is used when describing a MDDB database is because of the way the data is stored. From a conceptual standpoint, you can think of the data stored in an MDDB as being stored in multidimensional arrays, where we're usually dealing with much more than 2 dimensions. This is much different from a relational database, because in a relational database table you are really only working in two dimensions, namely, row and column. So if I have a table called **coder** in a relational database, and the columns in that table are handle and rating, the handle and rating columns represent a single dimension, and the actual rows of data in the table represent another single dimension (refer to figure 1).

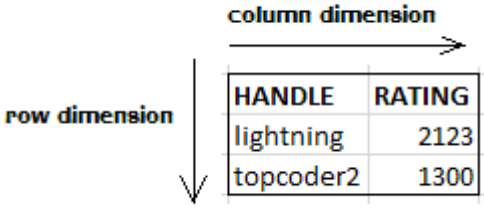


Figure 1

However, when we talk about cubes, we're typically dealing with many dimensions, not just two. When you design a cube, one of the first things you must do is determine the dimensions you want in the cube. It's very important to spend time really thinking through your cube **design** before implementing it, because if you don't choose your dimensions wisely it can cause the cube to perform poorly and, even worse, you won't meet the business needs. It's important to understand that the dimensions are really the architectural blueprint of your cube.

So going back to our sales warehouse example, we might have a cube with a Time dimension, a Scenarios dimension, and a Measures dimension (see Figure 2 below). The Scenarios dimension might contain members for plan and actual, which would correspond to planned (i.e. budgeted) values and actual (i.e. real sales) values respectively. Our Measures dimension could contain members for things like number of total orders taken, number of orders which were shipped or cancelled, and the order fulfillment rate, which is the ratio of orders shipped vs. orders placed. Finally, our Time dimension would contain members representing the fiscal calendar for the company, like fiscal week number, month of the year, fiscal quarter, etc. So conceptually, our cube might look like figure 2, shown below.

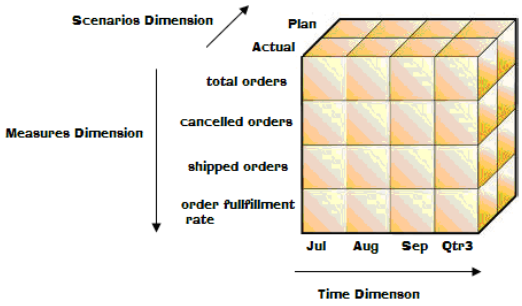


Figure 2

From a coder perspective, you could think of this cube as a three-dimensional array, in which the aforementioned dimension names correspond to dimensions in the array. And to extend that analogy further, if you wanted to know how many actual orders were shipped for September, you could look at cube[Actual][Sep][shipped orders], which would correspond to the "data cell" in the cube that contained that piece of data.

The real power of an OLAP cube stems from the fact that as we access those data cells in the multidimensional "array", the access time is almost instantaneous. Now, let's stop for a minute and think about how we would determine how many actual orders were shipped for September from a relational database. We would most likely write a SQL query to extract the answer from the relational database tables. However, as we mentioned earlier we obviously wouldn't want to run that type of query on our OLTP tables as that could slow down the performance of the OLTP database. But not only that, we would have to make sure that the relational database tables had the appropriate indexes on the tables we were querying so that the query would run in a reasonable amount of time. Remember, OLTP database tables can contain billions of records.

In summary, one of the primary advantages offered by MDDBs is the fast access they offer to data.

Dimensions

We introduced dimensions in the previous section, but now we'll take a more formal look at them. Dimensions, as their name imply, provide a way to categorize data into different groups. For example, let's pretend that we're building an OLAP cube to store TopCoder data about coders so we can analyze it. We need to think about the different types of data we have and see how we might categorize it, so let's make a list of some items in which we might be interested:

- Coder Handle
- Algorithm Rating
- Design Rating
- Development Rating
- Algorithm Rating Color
- Design Rating Color
- Development Rating Color
- Continent
- Country
- Join Date
- School
- Number of Algorithm Rated Events
- Number of SRM Wins
- Earnings

Next, we need to come up with some logical groupings for this data, and those logical groupings will become our dimensions. So let's start with the "measures", which are typically numeric values or percentages. We've already seen an example of a Measures dimension in the sales cube from the previous section. In the above list, we could make measures out of the numeric rating fields and the number of algorithm rated events field, so our measures dimension could look look like this:

Measures
  Algorithm Rating
  Development Rating
  Design Rating
  Number of Algorithm Rated Events
  Number of SRM Wins
  Earnings

The way to think about measures is to think of them as the "data values" you are storing in the multidimensional array. All the other dimensions are focused around analyzing these measures. In other words, measures are really a special dimension. Rating is clearly a measurement in the case of our coder cube, because it has a numeric value, and the same is true for the number of algorithm rated events, the number of SRM wins, and earnings.

Next, let's look at coder handle. We have some options here, and one of them would be to combine coder handle, country and geographic region into one dimension. In other words, within each dimension, we can have a hierarchy. So we could do something like this for our coder dimension:

Figure 3

Coder
  Asia
    China
      Coder1
      Coder2
      Coder3
      .
      .
    North Korea
      Coder4
      Coder5
      Coder6
    .
    .
  North America
    United States
      Coder7
      Coder8
      .
      .
    .
    .
  Europe
    France
      Coder9
      Coder10
      .
      .
    .
    .

In this example, we've started our hierarchy with the continent, and underneath each continent we have country, and underneath each country we have the actual coders for that country. What we have to decide here is whether this hierarchy will meet the needs of our users. The key thing to notice is that the more data you combine into a dimension, the less flexibility you have later. For example, what if we want to run some analysis by coder country? We can do it with this hierarchy, but it's not as easy as it could be if we had country as a separate dimension. In other words, to run analysis by country with the hierarchy above, we have to drill into each continent and find all the countries for that continent. If, on the other hand, we had country as its own dimension, then we could run the analysis by country much more easily since we wouldn't have to drill in by continent to find each country. For the purposes of our discussion, we'll assume that the above hierarchy is sufficient and we go with that for our coder dimension.

This brings us to another concept regarding cubes, namely, **aggregation**. The idea behind aggregation is that we typically load values to the lowest **level** in each dimension in the cube, and then sum up, or "aggregate" the values to the upper levels in each dimension. With Hyperion® Essbase, level 0 represents the leaf level members in each dimension. For example, in figure 3 above, the coder members of the dimension would be level 0. The parents of level 0 members are level 1 members, so the countries would be level 1. The parents of level 1 members are level 2, and we continue this numbering scheme on up until we get to the root member of the dimension. So when we are loading our cube with coder data, we would load it to the lowest level in our coder dimension. In other words, we would load it at the coder level. And after we load that data, we can then "aggregate" or roll it up to the upper levels. This is one of the main reasons OLAP cubes are so powerful. Since those values are aggregated, we don't have to do any summing when retrieving the values from the cube. The sum is already pre-calculated! You can think of a cube as sort of a "power array", in that we can look at any combination of members from the dimensions in the cube and get the answer in which we're interested. So conceptually, we could look at cube[Coder1][Earnings] to get Coder1's total earnings, or cube[China][Earnings] to get the total earnings for Chinese coders. And again, the real power here is that this answer is already pre-calculated in the OLAP cube so access time is minimal.

Again, let's contrast how we would get this information with a relational database. In a relational database, if you wanted to know the total earnings for coders from China, you would have to run a SQL statement and sum up the earnings (assuming that you hadn't built some kind of summary table in the relational database already). This may not seem like much when you're only talking about a few thousand coder records, and a SQL query would probably do just fine in this example. But if you consider having to run a SQL query to sum up values for 800 million transaction records from a sales database, you can see where OLAP cubes offer a huge advantage by having the values of interest already aggregated.

Some of you savvy folks out there might have noticed that aggregation doesn't necessarily make sense for all measures. For example, summing up two coders' ratings doesn't provide a meaningful value. So for these types of values, we **wouldn't** want to aggregate them to upper levels. Most MDDB vendors have a way to control this when you set up the cube. For example, with Hyperion® Essbase you can specify the type of aggregation to use, such as addition, subtraction, etc., but you can give the dimension member the "no consolidation" property, which means its values won't be rolled up to its parents.

So now that we've defined our coder and measures dimension, let's look at the other dimensions we need. We obviously want to store information about a coder's colors, and that information is in the algorithm rating color, development rating color, and design rating color values. We could make members in our measures dimension for these items, and just store red, yellow, etc. for the actual data value in the cube. And that may be fine, depending on the needs and types of analysis that are going to be performed on the cube. The other choice we could make here would be to make these fields into three cube dimensions, as shown below in figure 4:

Figure 4

Algorithm Rating Color
- Algo-Red
- Algo-Yellow
- Algo-Blue
- Algo-Green
- Algo-Gray
- Algo-Not Rated

Design Rating Color
- Design-Red
- Design-Yellow
- Design-Blue
- Design-Green
- Design-Gray
- Design-Not Rated

Development Rating Color
- Dev-Red
- Dev-Yellow
- Dev-Blue
- Dev-Green
- Dev-Gray
- Dev-Not Rated

Notice that the color in each dimension is prefixed with Algo, Design, or Dev respectively. The reason for this prefix is that with some MDDB vendors (depending on version), each member name must be unique across every dimension in the cube. Later versions of Hyperion® Essbase offer the ability to have duplicate member names, but earlier versions did not.

Now, what advantage does having the three rating color values as separate dimensions offer us over having them as members in the measures dimension? Well, it gives us the ability to "drill-in" to the cube to find the total earnings for all coders from North America who are Algo-Red and Dev-Blue, for example (i.e. cube[North America][Algo-Red][Dev-Blue]). Note that we could also drill into the "North America" member and find out the actual coders. The way this works is that when you drill into the cube, if no data is present for a member then it won't bring the member back (depending on the add-in settings). So if you drill down into North America, but there are no members that are both Algo-Red and Dev-Blue, then no members would be brought back. Different MDDB vendors have various tools for interfacing with a cube, but most of the more popular ones use some kind of Microsoft® Excel Add-In because spreadsheets work extremely well for working with cubes and drilling in and out of dimensions. With Hyperion® Essbase, for example, if we were to connect to the cube we have defined so far, we might be presented with a sheet like figure 5 below:

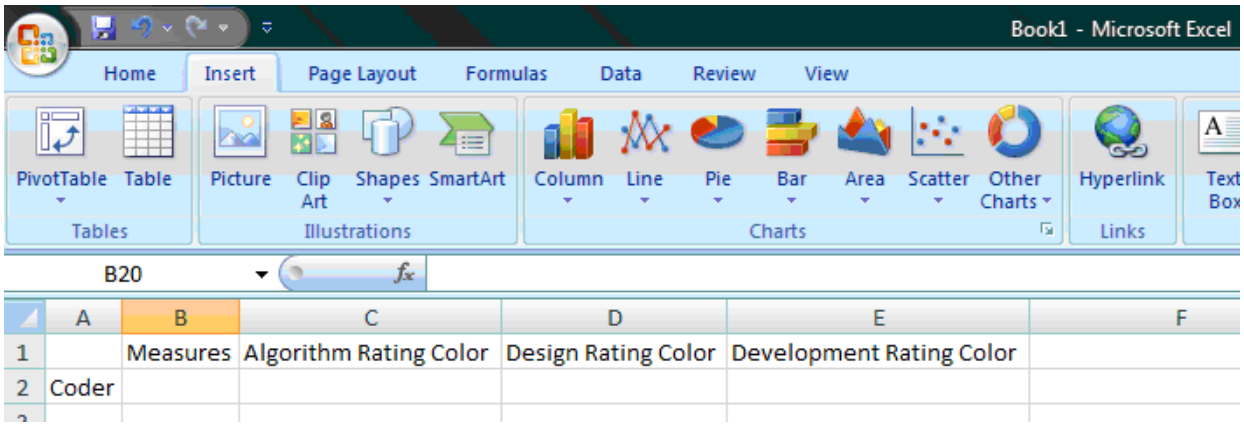


Figure 5

From there, you can "double-click" any dimension, and the add-in will drill into the dimension. So if I double-clicked on Coder, I'd end up something like this (obviously there would be many more countries but this is just for illustrative purposes):

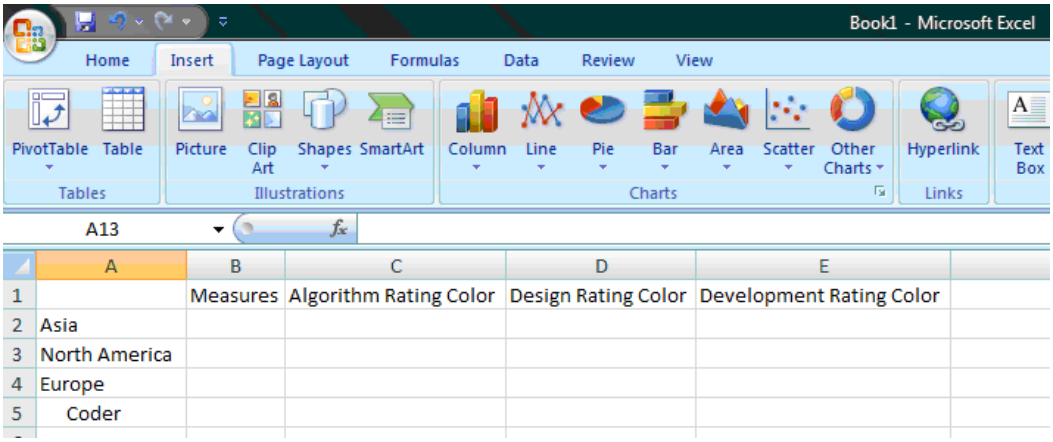


Figure 6

I could keep drilling into dimensions, and drill into the measures I'm interested in to see the actual values. So we could drill into Measures, and also navigate down to the Algo-Red, Design-Blue, Dev-Not Rated members and end up with this configuration:

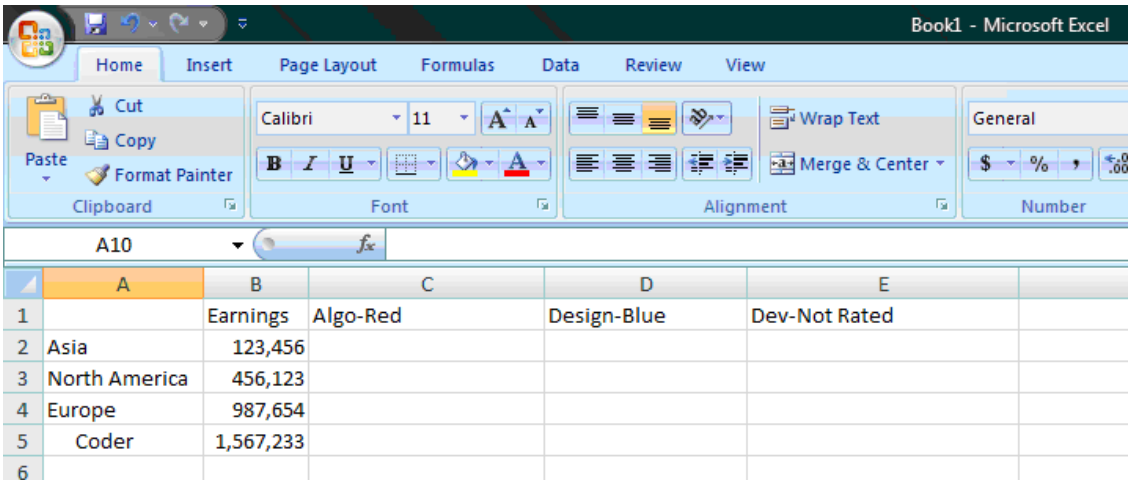


Figure 7 (illustrative purposes only, not actual data)

As you can see in figure 7, we now know the total earnings by continent for coders who are red in algorithms, blue in design, and not rated in development.

I hope you can see how this "drilling" ability gives the user virtually endless possibilities to slice and dice data into various formats to do the analysis they need. They can look at the dimensions at various levels (ex. Continent, country, etc.) to get the summary information in which they're interested. And the beauty of it all is that once you build a cube for them, they can do this type of analysis themselves and you rarely have to write custom reports (that use relational database data) for them anymore! It truly puts the power in the analyst's hands, and we'll look at a real world example of that a bit later.

We've made great progress on our cube! There are only two more values left to map, namely, school and join date. We'll go ahead and make school its own dimension since it probably makes sense to be able to do analysis at school level.

As for join date, usually with date values it's best to split date into two dimensions: one for year and one for calendar. We usually call the calendar dimension "Time". Here's an example:

Figure 8

Year
  2008
  2007
  2006
  2005
  2004
  2003
  2002
  2001
  2000
Time
  Qtr_1
    Jan
      Jan_1st
      Jan_2nd
      Jan_3rd
      .
      .
    Feb
      Feb_1st
      Feb_2nd
      Feb_3rd
      .
      .
    Mar Mar_1st
      Mar_2nd
      Mar_3rd
      .
      .
  Qtr_2
    Apr
      Apr_1st
      Apr_2nd
      Apr_3rd
      .
      .
    May
      May_1st
      May_2nd
      May_3rd
      .
      .
    Jun
      Jun_1st
      Jun_2nd
      Jun_3rd
      .
      .
  Qtr_3

.
.
Qtr_4
.
.

So we essentially create a time hierarchy so that if we want to know which coders joined in the first quarter of 2004, we can drill into the above dimensions and find that information. And by using this dimension structure, we don't ever have to change our Time dimension; we just add a new member to our Year dimension as we reach a new year, or we could just pre-add as many as we need.

Besides standard dimensions, another type of dimension you can use in a cube is an "attribute dimension". Different MDDB vendors implement attribute dimensions in various ways, but the way to think of attribute dimensions is that they are "characteristics" of members of another dimension. For example, if join date wasn't something that we needed to be able to do heavy analysis on, we could make it an attribute dimension associated with the coder dimension, since each coder has a join date. The thing to remember about attribute dimensions is that they don't give you quite the analysis power that a normal dimension does, so you have to ensure that they are going to meet your needs. For example, you can only associate one member of an attribute dimension with a member of a standard dimension. So let's say, for example, that we wanted to track coder's algo rating over time, but we wanted to make algo rating an attribute dimension. That would not work, because you could only have one rating value associated with a coder since it's an attribute dimension. In that case, you'd have to use two standard dimensions, one for rating time and one for rating. The other thing to remember about attribute dimensions is that the values aren't pre-aggregated in the cube like they are with standard dimensions, so the aggregation happens at data retrieval time. This can be a performance hit if you have many members in the attribute dimension, so you want to be smart in you considerations of where to use them or a standard dimension.

One word of caution about dimensions: the more dimensions you have, typically the longer it will take to aggregate the data and drill into dimensions. In other words, you don't want to just keep adding dimensions arbitrarily, because eventually if you get too many dimensions the cube is going to suffer performance-wise.

So at this point, we have designed our first OLAP cube! Give yourself a pat on the back!

Building Dimensions & Loading Data to the Cube

Obviously, a cube is worthless without data. We already alluded to ETL earlier in the article, and that is the way we get the data in the relational database into the form we need for loading into the cube. Sometimes, nothing has to be done here (i.e. the data can be loaded directly from the relational tables "as is" without any ETL). In either case, we must get the data into the cube. And note that many MDDB vendors don't restrict you to using a relational database as the data source for loading cube data. You can use spreadsheets, CSV files, and many other types of data source.

Before we can load the data into the cube, however, we must make sure we have all the dimension members present for the data which we wish to load. In other words, we can't load Coder14 data until we've actually added the Coder14 member to the coder dimension. Again, how you build dimensions varies with MDDB vendors. With Hyperion® Essbase, for example, you use what's known as a "load rule". With a load rule, you typically write an SQL statement (assuming you're using a relational database as the data source) to query for the dimension members you wish to load. You can load an individual dimension, or multiple dimensions in the same load rule. You can also specify if members are to be removed if they aren't found in the data source, so that is one way you can remove members from the cube (inactive coders, for example).

One the dimension members have been added, we're ready to load the actual cube data. The load rule for loading data to the cube must specify a member value for each dimension. If you think about it, that makes sense because if we leave out a dimension, then the loader won't know where to put the data in the cube. As mentioned earlier, it's usually best to load data to the lowest level of each dimension. Note that if you accidentally load a data value to a higher level, then when you "calculate" the cube (covered in the next section), you will wipe out the values because the aggregation will roll up the lower level value, which is nothing, to the higher level. As an example, if I loaded earnings values to the country level instead of the coder level, when I did the aggregation the country values would be overwritten because it would aggregate the values at coder level upwards.

Load rules are a big subject, and we've just touched on them briefly here. There are many caveats to deal with when loading data to a cube, such as what to do with rejected records, and how to handle duplicate member names. It's important to have a good strategy in place for handling these types of situations whenever you are going to build cubes. The strategy may vary from cube to cube, because with some cubes you may not care as much if records are rejected, but in other cubes it may be a critical event.

Calculations

So now that we've loaded our data to the OLAP cube, we need to aggregate it up to the higher levels. To do that, we must run a "calculation" step, which does just that. Note that we are not limited to just aggregating (i.e. rolling up) data, we can also have complex calculation formulas on data members. The industry standard for writing these types of calculations is MDX, which is a query language for OLAP databases. It's like SQL in a lot of ways, but it's sort of a paradigm shift because you have to think a bit differently when working with MDX because you are working in many dimensions, not just two.

MDX Calculation scripts can do other things as well, such as allocating a sales adjustment across several products over a time period. For example, let's say a department store has a 6 million dollar sales adjustment. This dollar amount needs to be distributed across fiscal year 2008, 1st quarter sales. The MDX script could take that 6 million dollars and "spread" it down to the lowest levels in each dimension, so that each product would contain a portion of the adjustment. These types of adjustments are quite common in sales cubes, where things like kickbacks and other types of dollar adjustments occur on a frequent basis (quarterly, for example) and have to be distributed across products.

MDX calculations also allow us to do simple calculations, such as creating a formula to add two values together and subtract another. For example, we might have the following formula for net sales:

Net_Sales = Gross_Sales - Cost of Goods Sold + Sales Adjustments

We could do that calculation ourselves in the load rule using a SQL expression, or we could let the cube calculate it for us. If you do the calculation in the SQL expression though, then you are aggregating computed values, which can cause a loss of precision as those computed values are aggregated to higher levels. So it's usually a good strategy to load the raw values and put the calculations in the cube itself.

Storage Options

In this section, we'll deal with some specifics regarding Hyperion® Essbase storage options because it's important to understand the options you have available. So far, we've been discussing the **Block Storage Option (BSO)**, which requires us to calculate the data to get it to higher levels. Obviously, this aggregation step can take substantial time if you have a lot of data and/or dimensions in the cube. One thing you can do with BSO cubes to reduce aggregation time is specify that a member is a **Dynamic Calc** member. This means that the value isn't calculated until retrieval time. The advantage here obviously is that your aggregation time is reduced. However, you must realize that it can slow down your retrieval time, because now the aggregation happens at fetch time. So it's a "pay me now, pay me later" situation. Generally, you want to limit dynamic calcs in a cube because you are usually going to load the cube in a nightly batch process, when time isn't a scarce resource. So a longer calc time doesn't matter as much since it's happening at night. But when an impatient financial analyst is fetching a value from the cube, you want that result to come back quickly.

However, Hyperion® Essbase also offers another type of storage option called **Aggregate Storage Option (ASO)**. With the ASO technology, no additional calculation step is required to aggregate the data! The values are aggregated dynamically at retrieval time. So basically, you load your data and you're done. ASO cubes are usually much smaller than their BSO counterparts, so that is one reason why the values can be aggregated at runtime and you can still get good retrieval performance. However, you can also pre-calculate values in an ASO cube and store them in aggregations, which is useful when the cube gets extremely big.

So which option should you choose, BSO or ASO? That depends. ASO cubes don't allow you to run calculation scripts, since all data retrieval is dynamic (except for the pre-calculated values we just mentioned). So in a financial cube, for example, where you are going to be doing a lot of allocation calculation scripts, BSO is probably the best choice.

However, in a cube where you want to have a lot of detailed data and want to be able to load and retrieve it quickly, ASO is a great option to consider.

Some Real World Examples

In this section, I'll share some real world examples of how Hyperion® Essbase OLAP technology has made a huge impact in one of the places I've worked.

In the shop where I worked, we were literally getting new report requests every 2-3 days for some new report for our forecasting system. The users wanted to see forecast totals by style, color, and size (it was an apparel manufacturing business), then somebody else would want to see totals by style only. Another user just wanted a report that showed total sales for last year for our "Big 3" retailers. In short, we couldn't finish

one report before there were 2 to 3 more user requests being added to our queue! That's when we decided to replace the entire forecasting system with an OLAP cube. One thing I haven't mentioned until now is that you can also load data to the cube through the Excel Add-In. So we made the OLAP cube the data repository for the forecast, so users would use the Excel Add-In to "lock and send" their forecast data to the cube using Excel. They also had the ability to calculate the cube (it was a BSO cube) so their forecast would be aggregated. We also loaded our actual sales data to the cube on a weekly basis so the users could compare actual to forecast to see how the company was doing, and also so they could plan the forecast accurately by looking at prior year sales.

Once we gave the forecast group this OLAP cube, amazingly, the weekly report requests suddenly stopped! We found that the users were able to meet all their reporting needs with the Excel Add-In and OLAP cube, and we no longer had to write custom reports. It was a huge win for the company, and the users absolutely loved it!

In another area of the company, we were having major problems with excess inventory and we needed a way to track and locate this inventory at the plants and get rid of it by selling it to our wholesale distributor. We had a short timeframe to do this, because after a selling season ended, the inventory was basically useless and the wholesaler would no longer buy it. So we developed an OLAP cube to track the inventory and compare it against the forecast (the same forecasting system I just described). Doing this allowed our financial planning group to see ahead of time which of our inventory was not going to be sold by subtracting forecast from inventory and seeing what was left. The financial planning department could then take a proactive approach to selling the inventory to our wholesaler before it became obsolete, and we literally saved millions of dollars in the process!

Conclusion

I hope by this point in the article you can see the advantages that MDDB's offer. The learning curve is a bit steep when you're getting acclimated with building cubes, but the payoff is tremendous, as you can use the technologies to make a huge impact in a company.

It seems that an almost obligatory and very important part of the recruitment process is "the test." "The test" can provide information both for the interviewer and the candidate. The interviewer is provided with a means to test the candidate's practical know-how and particular programming language understanding; the candidate can get an indication of the technologies used for the job and the level of proficiency the company expects and even decide if he still wants (and is ready for) the job in question.

I've had my fair share of interviews, more or less successful, and I would like to share with you my experience regarding some questions I had to face. I also asked for feedback from three of the top rated TopCoder members: bmerry, kyky and sql_lall , who were kind enough to correct me where I was not accurate (or plain wrong) and suggest some other questions (that I tried to answer to my best knowledge). Of course every question might have alternative answers that I did not manage to cover, but I tried to maintain the dynamics of an interview and to let you also dwell on them (certainly I'm not the man that knows all the answers). So, pencils up everyone and "let's identify potential C++ programmers or C++ programmers with potential."

1. **What is a class?**
   - A **class** is a way of encapsulating data, defining abstract data types along with initialization conditions and operations allowed on that data; a way of hiding the implementation (hiding the guts & exposing the skin); a way of sharing behavior and characteristics
2. **What are the differences between a C struct and a C++ struct?**
   - A C **struct** is just a way of combining data together; it only has characteristics (the data) and does not include behavior (functions may use the structure but are not tied up to it)
   - Typedefed names are not automatically generated for C structure tags; e.g.,:

```
// a C struct
struct my_struct {
   int someInt;
   char* someString;
};

// you declare a variable of type my_struct in C
struct my_struct someStructure;

// in C you have to typedef the name to easily
// declare the variable
typedef my_struct MyStruct;
MyStruct someOtherStuct;

// a C++ struct
struct MyCppStruct {
   int someInt;
   char* someString;
};

// you declare a variable of type MyCppStruct in C++
MyCppStruct someCppStruct;
// as you can see the name is automatically typedefed
```

   - But what's more important is that a C **struct** does not provide enablement for OOP concepts like encapsulation or polymorphism. Also **"C structs can't have static members or member functions"**, [bmerry]. A C++ **struct** is actually a **class**, the difference being that the default member and base class access specifiers are different: **class** defaults to private whereas **struct** defaults to public.
3. **What does the keyword const mean and what are its advantages over #define?**
   - In short and by far not complete, **const** means **"read-only"**! A named constant (declared with **const**) it's like a normal variable, except that its value cannot be changed. Any data type, user-defined or built-in, may be defined as a **const**, e.g.,:

```
// myInt is a constant (read-only) integer
const int myInt = 26;

// same as the above (just to illustrate const is
// right and also left associative)
int const myInt = 26;

// a pointer to a constant instance of custom
// type MyClass
const MyClass* myObject = new MyObject();

// a constant pointer to an instance of custom
```

- // type MyClass
- MyClass* const myObject = new MyObject();
-
- // myInt is a constant pointer to a constant integer
- const int someInt = 26;
  const int* const myInt = &someInt;

- **#define** is error prone as it is not enforced by the compiler like **const** is. It merely declares a substitution that the preprocessor will perform without any checks; that is **const** ensures the correct type is used, whereas **#define** does not. "Defines" are harder to debug as they are not placed in the symbol table.
- A constant has a scope in C++, just like a regular variable, as opposed to "defined" names that are globally available and may clash. A constant must also be defined at the point of declaration (must have a value) whereas "defines" can be "empty."
- Code that uses **const** is inherently protected by the compiler against inadvertent changes: e.g., to a class' internal state (**const** member variables cannot be altered, **const** member functions do not alter the class state); to parameters being used in methods (**const** arguments do not have their values changed within methods) [sql_lall]. A named constant is also subject for compiler optimizations.
- In conclusion, you will have fewer bugs and headaches by preferring **const** to **#define**.

4. **Can you explain the private, public and protected access specifiers?**
   - **public**: member variables and methods with this access specifier can be directly accessed from outside the class
   - **private**: member variables and methods with this access specifier cannot be directly accessed from outside the class
   - **protected**: member variables and methods with this access specifier cannot be directly accessed from outside the class with the exception of child classes
   - These access specifiers are also used in inheritance (that's a whole other story, see next question). You can inherit publicly, privately or protected (though I must confess, I cannot see the benefits of the latter).
5. **Could you explain public and private inheritance?[kyky, sql_lall]**
   - Public inheritance is the "default" inheritance mechanism in C++ and it is realized by specifying the public keyword before the base class
   - class B : public A
   - {
     };

   - Private inheritance is realized by specifying the private keyword before the base class or omitting it completely, as private is the default specifier in C++
   - class B : private A
   - {
   - };
   - or
   - class B : A
   - {
     };

   - The public keyword in the inheritance syntax means that the publicly/protected/privately accessible members inherited from the base class stay public/protected/private in the derived class; in other words, the members maintain their access specifiers. The private keyword in the inheritance syntax means that all the base class members, regardless of their access specifiers, become private in the derived class; in other words, private inheritance degrades the access of the base class' members - you won't be able to access public members of the base class through the derived one (in other languages, e.g., Java, the compiler won't let you do such a thing).
   - From the relationship between the base and derived class point of view,

     class B : public A {}; B **"is a"** A but class B : private A {};

     means B **"is implemented in terms of"** A.

   - Public inheritance creates subtypes of the base type. If we have class B : public A {}; then any B object is substituteable by its base calls object (through means of pointers and references) so you can safely write

     A* aPointer = new B();

     Private inheritance, on the other hand, class B : private A {};, does not create subtypes making the base type inaccessible and is a form of object composition. The following illustrates that:

```
class A
{
public:
   A();
   ~A();
   void doSomething();
};

void A :: doSomething()
{

}

class B : private A
{
public:
   B();
   ~B();
};
B* beePointer = new B();

// ERROR! compiler complains that the
// method is not accessible
beePointer->doSomething();
// ERROR! compiler complains that the
// conversion from B* to A* exists but
// is not accessible
A* aPointer = new B();
// ! for the following two the standard
// stipulates the behavior as undefined;
// the compiler should generate an error at least
// for the first one saying that B is not a
// polymorphic type
A* aPointer2 = dynamic_cast<A*>(beePointer);
A* aPointer3 = reinterpret_cast<A*>(beePointer);
```

6. **Is the "friend" keyword really your friend?[sql_lall]**
   - The **friend** keyword provides a convenient way to let specific nonmember functions or classes to access the private members of a class
   - friends are part of the class interface and may appear anywhere in the class (class access specifiers do not apply to friends); friends must be explicitly declared in the declaration of the class; e.g., :

```
class Friendly;

// class that will be the friend
class Friend
{
public:
   void doTheDew(Friendly& obj);
};

class Friendly
{
   // friends: class  and function; may appear
   // anywhere but it's
   // better to group them toghether;
   // the default private access specifier does
   // not affect friends
   friend class Friend;
   friend void friendAction(Friendly& obj);
public:
   Friendly(){ };
   ~Friendly(){ };
private:
   int friendlyInt;
```

```
};

// the methods in this class can access
// private members of the class that declared
// to accept this one as a friend
void Friend :: doTheDew(Friendly& obj) {
   obj.friendlyInt = 1;
}

// notice how the friend function is defined
// as any regular function
void friendAction(Friendly& obj)
{
   // access the private member
   if(1 == obj.friendlyInt)
   {
      obj.friendlyInt++;
   } else {
      obj.friendlyInt = 1;
   }
}
```

- "friendship isn't inherited, transitive or reciprocal," that is, your father's best friend isn't your friend; your best friend's friend isn't necessarily yours; if you consider me your friend, I do not necessarily consider you my friend.
- Friends provide some degree of freedom in a class' interface design. Also in some situations friends are syntactically better, e.g., operator overloading - binary infix arithmetic operators, a function that implements a set of calculations (same algorithm) for two related classes, depending on both (instead of duplicating the code, the function is declared a friend of both; classic example is Matrix * Vector multiplication).
- And to really answer the question, yes, **friend** keyword is indeed our friend but always "prefer member functions over nonmembers for operations that need access to the representation."[Stroustrup]

7. **For a class MyFancyClass { }; what default methods will the compiler generate?**
   - The default constructor with no parameters
   - The destructor
   - The copy constructor and the assignment operator
   - All of those generated methods will be generated with the **public** access specifier
   - E.g. MyFancyClass{ }; would be equivalent to the following :

```
class MyFancyClass
{
public:
   // default constructor
   MyFancyClass();
   // copy constructor
   MyFancyClass(const MyFancyClass&);
   // destructor
   ~MyFancyClass();


   // assignment operator
   MyFancyClass& operator=(const MyFancyClass&);
};
```

   - All of these methods are generated only if needed
   - The default copy constructor and assignment operator perform **memberwise** copy construction/assignment of the non-static data members of the class; if references or constant data members are involved in the definition of the class the assignment operator is not generated (you would have to define and declare your own, if you want your objects to be assignable)
   - I was living under the impression that the unary & (address of operator) is as any built-in operator - works for built-in types; why should the built-in operator know how to take the address of your home-brewed type? I thought that there's no coincidence that the "&" operator is also available for overloading (as are +, -, >, < etc.) and it's true is not so common to overload it, as you can live with the one generated by the compiler that looks like the following:

```
inline SomeClass* SomeClass::operator&()
{
   return this;
}
```

Thanks to bmerry for making me doubt what seemed the obvious. I found out the following:

From the ISO C++ standard:
Clause 13.5/6 [over.oper] states that operator =, (unary) & and , (comma) have a predefined meaning for each type. Clause 5.3.1/2 [expr.unary.op] describes the meaning of the address-of operator. No special provisions are mode for class-type objects (unlike in the description of the assignment expression). Clause 12/1 [special] lists all the special member functions, and states that these will be implicitly declared if needed. The address-of operator is not in the list.
From Stroustrup's The C++ Programming Language - Special 3rd Edition:
"Because of historical accident, the operators = (assignment), & (address-of), and , (sequencing) have predefined meanings when applied to class objects. These predefined meanings can be made inaccessible to general users by making them private:... Alternatively, they can be given new meanings by suitable definitions."
From the second edition of Meyer's Effective C++:
"A class declaring no operator& function(s) does NOT have them implicitly declared. Rather, compilers use the built-in address-of operator whenever "&" is applied to an object of that type. This behavior, in turn, is technically not an application of a global operator& function. Rather, it is a use of a built-in operator." In the errata http://www.aristeia.com/BookErrata/ec++2e-errata_frames.html

8. **How can you force the compiler not to generate the above mentioned methods?**
   - Declare and define them yourself - the ones that make sense in your class' context. The default no-parameters constructor will not be generated if the class has at least one constructor with parameters declared and defined.
   - Declare them private - disallow calls from the outside of the class and DO NOT define them (do not provide method bodies for them) - disallow calls from member and friend functions; such a call will generate a linker error.
9. **What is a constructor initialization list?**
   - A **special** initialization point in a constructor of a class (initially developed for use in inheritance).
   - Occurs only in the definition of the constructor and is a list of constructor calls separated by commas.
   - The initialization the constructor initialization list performs occurs before any of the constructor's code is executed - very important point, as you'll have access to fully constructed member variables in the constructor!
   - For example:

```
// a simple base class just for illustration purposes
class SimpleBase
{
public:
   SimpleBase(string&);
   ~SimpleBase();
private:
   string& m_name;
};

// example of initializer list with a call to the
// data member constructor
SimpleBase :: SimpleBase(string& name) : m_name(name)
{

}

// a class publicly derived from SimpleBase just for
// illustration purposes
class MoreComplex : public SimpleBase
{
public:
   MoreComplex(string&, vector<int>*, long);
   ~MoreComplex();
private:
   vector<int>* m_data;
   const long m_counter;
};


// example of initializer list with calls to the base
// class constructor and data member constructor;
// you can see that built-in types can also be
// constructed here
MoreComplex :: MoreComplex(string &name,
```

```
    vector<int>* someData, long counter) :
    SimpleBase(name), m_data(someData),
    m_counter(counter)
{

}
```

- As you saw in the above example, built-in types can also be constructed as part of the constructor initialization list.
- Of course you do not have to use the initialization list all the time (see the next question for situations that absolutely require an initialization list) and there are situations that are not suitable for that: e.g., you have to test one of the constructor's arguments before assigning it to your internal member variable and throw if not appropriate.
- It is recommended that the initialization list has a consistent form: first the call to the base class(es) constructor(s), and then calls to constructors of data members in the order they were specified in the class' declaration . Note that this is just a matter of coding style: you declare your member variables in a certain order and it will look good and consistent to initialize them in the same order in the initialization list.

10. **When "must" you use a constructor initialization list?**
    - **Constant** and **reference** data members of a class may only be initialized, never assigned, so you **must** use a constructor initialization list to properly construct (initialize) those members.
    - In inheritance, when the base class does not have a default constructor or you want to change a default argument in a default constructor, you have to explicitly call the base class' constructor in the initialization list.
    - For reasons of correctness - any calls you make to member functions of sub-objects (used in composition) go to initialized objects.
    - For reasons of efficiency. Looking at the previous question example we could rewrite the SimpleBase constructor as follows:

```
SimpleBase :: SimpleBase(string &name)
{
    m_name = name;
}
```

The above will generate a call to the default string constructor to construct the class member m_name and then the assignment operator of the string class to assign the name argument to the m_name member. So you will end up with two calls before the data member m_name is fully constructed and initialized.

```
SimpleBase :: SimpleBase(string &name) : m_name(name)
{

}
```

The above will only generate a single call, which is to the copy constructor of the string class, thus being more efficient.

In the second part of this installment we'll tackle some questions regarding more advanced features of the language (the experienced C++ programmers will consider some of these more on the basic side). So let's get to it and work on the second part of this "interview".

1. **What are virtual functions?**
    - Virtual functions represent the mechanism through which C++ implements the OO concept of polymorphism. Virtual functions allow the programmer to redefine in each derived class functions from the base class with altered behavior so that you can call the right function for the right object (allow to perform the right operation for an object through only a pointer/reference to that object's base class)
    - A member function is declared virtual by preceding its declaration (not the definition) with the **virtual** keyword

```
class Shape
{
public:
    ...
    //a shape can be drawn in many ways
    virtual void draw(){ };
};
```

"A virtual function must be defined for the class in which it is first declared ..." [Stroustrup]. The redefinition of a virtual function in a derived class is called **overriding** (complete rewrite) or **augmentation** (rewrite but with a call to the base class function)

```
class Rectangle : public Shape
{
```

```
public:
   ...
   void draw() { };
};

class Square : public Rectangle
{
public:
   ...
   void draw() { };
};
   ...

Shape* theShape = new Square();
// with the help of virtual functions
// a Square will be drawn and not
// a Rectangle or any other Shape
theShape->draw();
```

- Through virtual functions C++ achieves what is called **late binding** (dynamic binding or runtime binding), that is actually connecting a function call to a function body at runtime based on the type of the object and not at compilation time (static binding) **(**)**

2. **What is a virtual destructor and when would you use one?**
   - A virtual destructor is a class' destructor conforming to the C++'s polymorphism mechanism; by declaring the destructor virtual you ensure it is placed in the **VTABLE** of the class and it will be called at proper times
   - You make a class' destructor virtual to ensure proper clean-up when the class is supposed to be subclassed to form a hierarchy and you want to delete a derived object thorough a pointer to it (the base class)
   - E.g. :

```
#include <vector>
#include <iostream>

using namespace std;

class Base
{
public:
   Base(const char* name);
   // warning! the destructor should be virtual
   ~Base();

   virtual void doStuff();
private:
   const char* m_name;
};

Base :: Base(const char* name) : m_name(name)
{

}

Base :: ~Base()
{

}



void Base :: doStuff()
{
   cout << "Doing stuff in Base" << endl;
}
```

```
class Derived : public Base
{
public:
   Derived(const char* name);
   ~Derived();

virtual void doStuff();
private:
   vector<int>* m_charCodes;
};

Derived :: Derived(const char* name) : Base(name)
{
   m_charCodes = new vector<int>;
}

Derived :: ~Derived()
{
   delete m_charCodes;
}

void Derived :: doStuff()
{
   cout << "Doing stuff in Derived" << endl;
}

int main(int argc, char* argv[])
{
   // assign the derived class object pointer to
   // the base class pointer
   char* theName = "Some fancy name";
   Base* b = new Derived(theName);

   // do some computations and then delete the
   // pointer
   delete b;
   return 0;
}
```

What will happen in our rather lengthy example? Everything seems OK and most of the available C++ compilers will not complain about anything **(*)**. Nevertheless there is something pretty wrong here. The C++ standard is clear on this topic: **when you want to delete a derived class object through a base class pointer and the destructor of the base class is not virtual the result is undefined**. That means you're on your own from there and the compiler won't help you! What is the most often behavior in such situations is that the derived class' destructor is never called and parts of your derived object are left undestroyed. In the example above you will leave behind a memory leak, the m_charCodes member will not be destroyed because the destructor ~Derived() will not be called

- A thing to notice is that declaring all destructors virtual is also pretty inefficient and not advisable. That makes sense (declaring the destructor virtual) only if your class is supposed to be part of a hierarchy as a base class, otherwise you'll just waste memory with the class' **vtable** generated only for the destructor. So declare a virtual destructor in a class **"if and only if that class is part of a class hierarchy, containing at least one virtual function. In other words, it is not necessary for the class itself to have that virtual function - it is sufficient for one of its descendents to have one."**[kyky]

3. **How do you implement something along the lines of Java interfaces in C++?**[kyky]
   - C++ as a language does not support the concept of "interfaces" (as opposed to other languages like Java or D for example), but it achieves something similar through **Abstract Classes**
   - You obtain an abstract class in C++ by declaring at least one **pure virtual function** in that class. A virtual function is transformed in a pure virtual with the help of the initializer **"= 0"**. A pure virtual function does not need a definition. An abstract class cannot be instantiated but only used as a base in a hierarchy

```
class MySillyAbstract
{
public:
   // just declared not defined
   virtual void beSilly() = 0;
```

```
};
```

A derivation from an abstract class must implement all the pure virtuals, otherwise it transforms itself into an abstract class

- You can obtain an **"interface"** in C++ by declaring an abstract class with all the functions pure virtual functions and public and no member variables - only behavior and no data

```
class IOInterface
{
public:
   virtual int open(int opt) = 0;
   virtual int close(int opt) = 0;
   virtual int read(char* p, int n) = 0;
   virtual int write(const char* p, int n) = 0;
};
```

[adapted after an example found in Stroustup The C++ Programming Language 3rd Edition]
In this way you can specify and manipulate a variety of IO devices through the interface.

4. **Could you point out some differences between pointers and references?**
   - A reference must always be initialized because the object it refers to already exists; a pointer can be left uninitialized (though is not recommended)
   - There's no such thing a s a **"NULL reference"** because a reference must always refer to some object, so the "no object" concept makes no sense; pointers, as we all know, can be **NULL**
   - References can be more efficient, because there's no need to test the validity of a reference before using it (see above comment); pointers most often have to be tested against **NULL** to ensure there are valid objects behind them
   - Pointers may be reassigned to point to different objects (except constant pointers, of course), but references cannot be (references are "like" constant pointers that are automatically dereferenced by the compiler)
   - References are tied to someone else's storage (memory) while pointers have their own storage they account for
   - One would use the dot operator **"."** to access members of references to objects, however to access members of pointers to objects one uses the arrow **"->"[sql_lall]**
5. **When would you use a reference?**
   - You should use a **reference** when you certainly know you have something to refer to, when you never want to refer to anything else and when implementing operators whose syntactic requirements make the use of pointers undesirable; in all other cases, "stick with pointers"
   - Do not use references just to reduce typing. That (and that being the sole reason) is not an appropriate usage of the reference concept in C++; using references having in mind just the reason of reduced typing would lead you to a "reference spree" - it must be clear in one's mind when to use references and when to use pointers; overusing any of the two is an inefficient path
6. **Can you point out some differences between new & malloc?**
   - **"new"** is an operator built-in into the C++ language, **"malloc"** is a function of the C standard library
   - **"new"** is aware of constructors/destructors, **"malloc"** is not; e.g. :

```
string* array1 = static_cast<string*>(malloc(10 * sizeof(string)));
free(array1);
```

array1 in the above example points to enough memory to hold 10 strings but no objects have been constructed and there's no easy and clean (proper) way from OO point of view to initialize them (see the question about **placement new** - in most day to day programming tasks there's no need to use such techniques). The call to **free()** deallocates the memory but does not destroy the objects (supposing you managed to initialize them).

```
string* array2 = new string[10];
delete[] array2;
```

on the other hand **array2** points to 10 fully constructed objects (they have not been properly initialized but they are constructed), because **"new"** allocates memory and also calls the **string** default constructor for each object. The call to the **delete** operator deallocates the memory and also destroys the objects

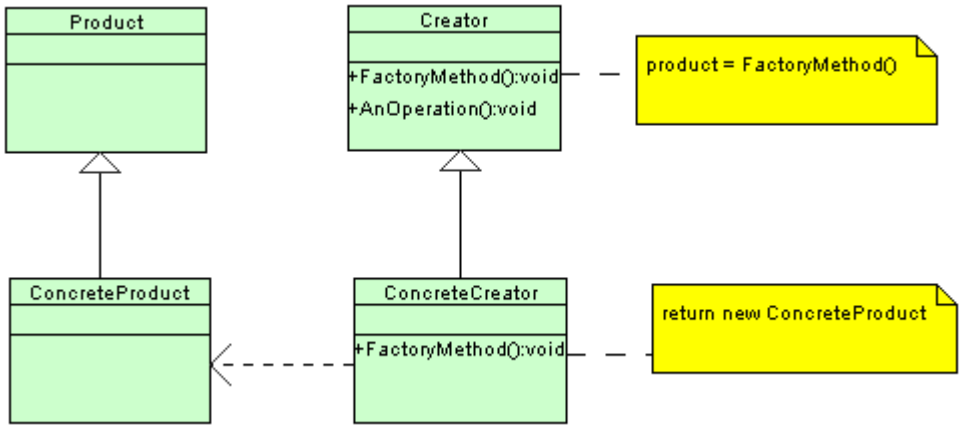
   - You got to remember to always use **free()** to release memory allocated with **malloc()** and **delete** (or the array correspondent **delete[]**) to release memory allocated with **new** (or the array correspondent **new[]**)
7. **What are the differences between "operator new" and the "new" operator?**
   - **"new"** is an operator built into the language and it's meaning cannot be changed; **"operator new"** is a function and manages **how** the **"new"** operator allocates memory its signature being: **void* operator new(size_t size)**

- The **"new"** operator is allowed to call a constructor, because **new** has 2 major steps in achieving its goals : in step 1 it allocates enough memory using **"operator new"** and then in step 2 calls the constructor(s) to construct the object(s) in the memory that was allocated
- **"operator new"** can be overridden meaning that you can change the way the **"new"** operator allocates memory, that is the mechanism, but not the way the **"new"** operator behaves, that is it's policy(semantics) , because what **"new"** does is fixed by the language

8. **What is "placement new"?**
   - A special form of constructing an object in a given allocated zone of memory
   - The caller already knows what the pointer to the memory should be, because it knows where is supposed to be placed. **"placement new"** returns the pointer that's passed into it
   - Usage of **"placement new"** implies an explicit call to the object's destructor when the object is to be deleted, because the memory was allocated/obtained by other means than the standard **"new"** operator allocation
   - E.g. :

```
// supposing a "buffer" of memory large enough for
// the object we want to construct was
// previously allocated using malloc
MyClass* myObject = new (buffer) MyClass(string& name);


// !!ERROR
delete myObject;
// the correct way is
myObject->~MyClass();
// then the "buffer" must also be properly
// deallocated
free(buffer);
```

9. **What is a "virtual constructor"?[kyky]**
   - There is no such thing as a virtual constructor in C++ simply because you need to know the exact type of the object you want to create and virtual represent the exact opposite concept **(***)**
   - But using an indirect way to create objects represents what is known as **"Virtual Constructor Idiom"**. For example you could implement a **clone()** function as an indirect copy constructor or a **create()** member function as an indirect default constructor (C++ FAQ Lite)
   - The GoF calls a variant of this idiom the Factory Method Pattern - "define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses". A concrete example will speak for itself:



[Created using the TopCoder UML Tool]

```
// Product
class Page
{
};

// ConcreteProduct
class SkillsPage : public Page
{
};
```

```
// ConcreteProduct
class ExperiencePage : public Page
{
};

// ConcreteProduct
class IntroductionPage : public Page
{
};

// ConcreteProduct
class TableOfContentsPage : public Page
{
};

// Creator
class Document
{
// Constructor calls abstract Factory method
public:
   Document();

   // Factory Method
   virtual void CreatePages() { };
protected:
   std::list<Page*> thePageList;
};

Document :: Document()
{
   CreatePages();
};

// ConcreteCreator
class Resume : public Document
{
public:
   // Factory Method implementation
   void CreatePages();
};

// Factory Method implementation
void Resume :: CreatePages()
{
   thePageList.push_back(new SkillsPage());
   thePageList.push_back(new ExperiencePage());
}

// ConcreteCreator
class Report : public Document
{
public:
   // Factory Method implementation
   void CreatePages();
};

// Factory Method implementation
void Report :: CreatePages()
{
   thePageList.push_back(new TableOfContentsPage());
   thePageList.push_back(new IntroductionPage());
}
```

```
int main(int argc, char* argv[])
{
   // Note: constructors call Factory Method
   vector<Document*> documents(2);
   documents[0] = new Resume();
   documents[1] = new Report();

   return 0;
}
```

10. **What is RAII?**
    - **RAII** - **R**esource **A**cquisition **I**s **I**nitialization - is a C++ technique (but not limited to the C++ language) that combines acquisition and release of resources with initialization and uninitialization of variables
    - E.g. :

```
// this is a hypothetic LogFile class using an
// hypothetic File class just for the illustration
// of the technique
class LogFile
{
public:
   LogFile(const char*);
   ~LogFile();

   void write(const char*);
private:
   File* m_file;
};

LogFile :: LogFile(const char* fileName) :
// ! acquisition and initialization
m_file(OpenFile(fileName))
{
   if(NULL == m_file)
   {
      throw FailedOpenException();
   }
}

LogFile :: ~LogFile()
{
   // ! release and uninitialization
   CloseFile(m_file);
}

void LogFile :: write(const char* logLine)
{
   WriteFile(m_file, logLine);
}

// a hypothetical usage example
void SomeClass :: someMethod()
{
   LogFile log("log.tx");
   log.write("I've been logged!");

   // !exceptions can be thrown without
   // worrying about closing the log file
   // or leaking the file resource
   if(...)
   {
   throw SomeException();
   }
}
```

- Without **RAII** each usage of the LogFile class would be also combined with the explicit management of the File resource. Also in the presence of exceptions you would have to be careful and clean-up after yourself, thing that is taken care of with the proper usage of **RAII** as illustrated in the example above
- **RAII** is best used with languages that call the destructor for local objects when they go out of scope (implicit support of the technique) like C++. In other languages, like Java & C#, that rely on the garbage collector to destruct local objects, you need finalization routines (e.g. try-finally blocks) to properly use **RAII**
- Real usage examples of the technique are the C++ Standard Library's file streams classes and STL's auto_ptr class (to name just very, very few)

That was it, folks! I hope that even if those questions did not pose any challenges, you still had fun doing/reading this quiz and refreshing your memory on some aspects of the C++ language. Good luck on those interviews!

Notes

(*) bmerry suggested that my claim is not accurate but I've tested the example on Windows XP: Visual Studio 2005 Professional Edition (the evaluation one that you can get from the Microsoft site) did not warn, not even after setting the warnings level to Level 4 (Level 3 is the default one); Mingw compiler based on GCC (that comes with the Bloodshed DevCpp version 4.9.9.2) also did not warn (the compiler settings from within the IDE are minimalist; tried to pass -pedantic and -Wextra to the compiler command line but still no success); Digital Mars C++ compiler (dmc) also did not warn with all warnings turned on; Code Warrior Professional Edition 9 does not warn also (this is pretty old, but Metrowerks compilers were renowned for the robustness and standard conformance). So, unless you start digging through the documentation of those compilers to find that right command line switch or start writing the right code, you're in the harms way at least with the "out of the box" installations of these compilers.

(**) The compiler does all the magic: first, for each class that contains virtual functions (base and derived), the compiler creates a static table called the **VTABLE**. Each virtual function will have a corresponding entry in that table (a function pointer); for the derived classes the entries will contain the overridden virtual functions' pointers. For **each base** class (it's not static, each object will have it) the compiler adds a hidden pointer called the **VPTR**, that will be initialized to point to the beginning of the **VTABLE** - in the derived classes the (same) **VPTR** will be initialized to point to the beginning of the derived class' **VTABLE**. So when "you make a virtual function call through a base class pointer (that is, when you make a polymorphic call), the compiler quietly inserts code to fetch the **VPTR** and look up the function address in the **VTABLE**, thus calling the correct function". This might seem overly complicated but on a typical machine it does not take much space and it's very, very fast as a smart man said once "fetch, fetch call".

(***) For that and other fine C++ gems go to Stroustrup.

Introduction

Primality testing of a number is perhaps the most common problem concerning number theory that topcoders deal with. A prime number is a natural number which has exactly two distinct natural number divisors: 1 and itself. Some basic algorithms and details regarding primality testing and factorization can be found here.

The problem of detecting whether a given number is a prime number has been studied extensively but nonetheless, it turns out that all the deterministic algorithms for this problem are too slow to be used in real life situations and the better ones amongst them are tedious to code. But, there are some probabilistic methods which are very fast and very easy to code. Moreover, the probability of getting a wrong result with these algorithms is so low that it can be neglected in normal situations.

This article discusses some of the popular probabilistic methods such as Fermat's test, Rabin-Miller test, Solovay-Strassen test.

Modular Exponentiation

All the algorithms which we are going to discuss will require you to efficiently compute $(a^b)\%c$ (where a,b,c are non-negative integers). A straightforward algorithm to do the task can be to iteratively multiply the result with 'a' and take the remainder with 'c' at each step.

```
/* a function to compute (a^b)%c */
int modulo(int a,int b,int c){
    // res is kept as long long because intermediate results might overflow in "int"
    long long res = 1;
    for(int i=0;i<b;i++){
        res *= a;
        res %= c; // this step is valid because (a*b)%c = ((a%c)*(b%c))%c
    }
    return res%c;
}
```

However, as you can clearly see, this algorithm takes O(b) time and is not very useful in practice. We can do it in O(log(b)) by using what is called as exponentiation by squaring. The idea is very simple:

$$a^b = \begin{cases} (a^2)^{(b/2)} & \text{if b is even and b > 0} \\ a*(a^2)^{((b-1)/2)} & \text{if b is odd} \\ 1 & \text{if b = 0} \end{cases}$$

This idea can be implemented very easily as shown below:

```
/* This function calculates (a^b)%c */
int modulo(int a,int b,int c){
    long long x=1,y=a; // long long is taken to avoid overflow of intermediate results
    while(b > 0){
        if(b%2 == 1){
            x=(x*y)%c;
        }
        y = (y*y)%c; // squaring the base
        b /= 2;
    }
    return x%c;
}
```

Notice that after i iterations, b becomes $b/(2^i)$, and y becomes $(y^{(2i)})\%c$. Multiplying x with y is equivalent to adding 2^i to the overall power. We do this if the i^{th} bit from right in the binary representation of b is 1. Let us take an example by computing $(7^{107})\%9$. If we use the above code, the variables after each iteration of the loop would look like this: (a = 7, c = 9)

iterations	b	x	y
0	107	1	7
1	53	7	4
2	26	1	7
3	13	1	4
4	6	4	7
5	3	4	4

```
  6   1  7   7
  7   0     4   4
```

Now b becomes 0 and the return value of the function is 4. Hence $(7^{107})\%9 = 4$.

The above code could only work for a,b,c in the range of type "int" or the intermediate results will run out of the range of "long long". To write a function for numbers up to 10^18, we need to compute (a*b)%c when computing a*b directly can grow larger than what a long long can handle. We can use a similar idea to do that:

```
   (2*a)*(b/2)           if b is even and b > 0
a*b = a + (2*a)*((b-1)/2)    if b is odd
   0      if b = 0
```

Here is some code which uses the idea described above (you can notice that its the same code as exponentiation, just changing a couple of lines):

```
/* this function calculates (a*b)%c taking into account that a*b might overflow */
long long mulmod(long long a,long long b,long long c){
    long long x = 0,y=a%c;
    while(b > 0){
        if(b%2 == 1){
            x = (x+y)%c;
        }
        y = (y*2)%c;
        b /= 2;
    }
    return x%c;
}
```

We could replace x=(x*y)%c with x = mulmod(x,y,c) and y = (y*y)%c with y = mulmod(y,y,c) in the original function for calculating $(a^b)\%c$. This function requires that 2*c should be in the range of long long. For numbers larger than this, we could write our own BigInt class (java has an inbuilt one) with addition, multiplication and modulus operations and use them.

This method for exponentiation could be further improved by using Montgomery Multiplication. Montgomery Multiplication algorithm is a quick method to compute (a*b)%c, but since it requires some pre-processing, it doesn't help much if you are just going to compute one modular multiplication. But while doing exponentiation, we need to do the pre-processing for 'c' just once, that makes it a better choice if you are expecting very high speed. You can read about it at the links mentioned in the reference section.

Similar technique can be used to compute $(a^b)\%c$ in $O(n^3 * \log(b))$, where a is a square matrix of size n x n. All we need to do in this case is manipulate all the operations as matrix operations. Matrix exponentiation is a very handy tool for your algorithm library and you can see problems involving this every now and then.

Fermat Primality Test

Fermat's Little Theorem
According to Fermat's Little Theorem if p is a prime number and a is a positive integer less than p, then

$a^p = a \pmod p$
or alternatively:
$a^{(p-1)} = 1 \pmod p$

Algorithm of the test
If p is the number which we want to test for primality, then we could randomly choose a, such that a < p and then calculate $(a^{(p-1)})\%p$. If the result is not 1, then by Fermat's Little Theorem p cannot be prime. What if that is not the case? We can choose another a and then do the same test again. We could stop after some number of iterations and if the result is always 1 in each of them, then we can state with very high probability that p is prime. The more iterations we do, the higher is the probability that our result is correct. You can notice that if the method returns composite, then the number is sure to be composite, otherwise it will be probably prime.

Given below is a simple function implementing Fermat's primality test:

/* Fermat's test for checking primality, the more iterations the more is accuracy */

```
bool Fermat(long long p,int iterations){
    if(p == 1){ // 1 isn't prime
        return false;
    }
    for(int i=0;i<iterations;i++){
        // choose a random integer between 1 and p-1 ( inclusive )
        long long a = rand()%(p-1)+1;
        // modulo is the function we developed above for modular exponentiation.
        if(modulo(a,p-1,p) != 1){
            return false; /* p is definitely composite */
        }
    }
    return true; /* p is probably prime */
}
```

More iterations of the function will result in higher accuracy, but will take more time. You can choose the number of iterations depending upon the application.

Though Fermat is highly accurate in practice there are certain composite numbers p known as Carmichael numbers for which all values of a<p for which gcd(a,p)=1, $(a^{(p-1)})\%p = 1$. If we apply Fermat's test on a Carmichael number the probability of choosing an a such that gcd(a,p) != 1 is very low (based on the nature of Carmichael numbers), and in that case, the Fermat's test will return a wrong result with very high probability. Although Carmichael numbers are very rare (there are about 250,000 of them less than 10^{16}), but that by no way means that the result you get is always correct. Someone could easily challenge you if you were to use Fermat's test :). Out of the Carmichael numbers less than 10^{16}, about 95% of them are divisible by primes < 1000. This suggests that apart from applying Fermat's test, you may also test the number for divisibility with small prime numbers and this will further reduce the probability of failing. However, there are other improved primality tests which don't have this flaw as Fermat's. We will discuss some of them now.

Miller-Rabin Primality Test

Key Ideas and Concepts

1. Fermat's Little Theorem.
2. If p is prime and $x^2 = 1$ (mod p), then x = +1 or -1 (mod p). We could prove this as follows:
3. $x^2 = 1$ (mod p)
4. $x^2 - 1 = 0$ (mod p)
5. (x-1)(x+1) = 0 (mod p)

Now if p does not divide both (x-1) and (x+1) and it divides their product, then it cannot be a prime, which is a contradiction. Hence, p will either divide (x-1) or it will divide (x+1), so x = +1 or -1 (mod p).

Let us assume that $p - 1 = 2^d * s$ where s is odd and d >= 0. If p is prime, then either $a^s = 1$ (mod p) as in this case, repeated squaring from a^s will always yield 1, so $(a^{(p-1)})\%p$ will be 1; or $a^{(s*(2r))} = -1$ (mod p) for some r such that 0 <= r < d, as repeated squaring from it will always yield 1 and finally $a^{(p-1)} = 1$ (mod p). If none of these hold true, $a^{(p-1)}$ will not be 1 for any prime number a (otherwise there will be a contradiction with fact #2).

Algorithm

Let p be the given number which we have to test for primality. First we rewrite p-1 as $(2^d)*s$. Now we pick some a in range [1,n-1] and then check whether $a^s = 1$ (mod p) or $a^{(s*(2r))} = -1$ (mod p). If both of them fail, then p is definitely composite. Otherwise p is probably prime. We can choose another a and repeat the same test. We can stop after some fixed number of iterations and claim that either p is definitely composite, or it is probably prime.

A small procedure realizing the above algorithm is given below:

```
/* Miller-Rabin primality test, iteration signifies the accuracy of the test */
bool Miller(long long p,int iteration){
    if(p<2){
        return false;
    }
    if(p!=2 && p%2==0){
        return false;
    }
    long long s=p-1;
```

```
    while(s%2==0){
        s/=2;
    }
    for(int i=0;i<iteration;i++){
        long long a=rand()%(p-1)+1,temp=s;
        long long mod=modulo(a,temp,p);
        while(temp!=p-1 && mod!=1 && mod!=p-1){
            mod=mulmod(mod,mod,p);
            temp *= 2;
        }
        if(mod!=p-1 && temp%2==0){
            return false;
        }
    }
    return true;
}
```

It can be shown that for any composite number p, at least (3/4) of the numbers less than p will witness p to be composite when chosen as 'a' in the above test. Which means that if we do 1 iteration, probability that a composite number is returned as prime is (1/4). With k iterations the probability of test failing is $(1/4)^k$ or $4^{(-k)}$. This test is comparatively slower compared to Fermat's test but it doesn't break down for any specific composite numbers and 18-20 iterations is a quite good choice for most applications.

Solovay-Strassen Primality Test

Key Ideas and Concepts

1. Legendre Symbol: This symbol is defined for a pair of integers a and p such that p is prime. It is denoted by (a/p) and calculated as:
2. = 0 if a%p = 0
3. (a/p) = 1 if there exists an integer k such that $k^2 = a \pmod p$
   = -1 otherwise.

   It is proved by Euler that:

   $(a/p) = (a^{((p-1)/2)}) \% p$

   So we can also say that:

   $(ab/p) = (ab^{((p-1)/2)}) \% p = (a^{((p-1)/2)})\%p * (b^{((p-1)/2)})\%p = (a/p)*(b/p)$

4. Jacobian Symbol: This symbol is a generalization of Legendre Symbol as it does not require 'p' to be prime. Let a and n be two positive integers, and $n = p1^{k1} * .. * pn^{kn}$, then Jacobian symbol is defined as:
5. $(a/n) = ((a/p1)^{k1}) * ((a/p2)^{k2}) * ..... * ((a/pn)^{kn})$

   So you can see that if n is prime, the Jacobian symbol and Legendre symbol are equal.

   There are some properties of these symbols which we can exploit to quickly calculate them:

   1. (a/n) = 0 if gcd(a,n) != 1, Hence (0/n) = 0. This is because if gcd(a,n) != 1, then there must be some prime pi such that pi divides both a and n. In that case (a/pi) = 0 [by definition of Legendre Symbol].
   2. (ab/n) = (a/n) * (b/n). It can be easily derived from the fact (ab/p) = (a/p)(b/p) (here (a/p) is the Legendry Symbol).
   3. if a is even, than (a/n) = (2/n)*((a/2)/n). It can be shown that:
   4. = 1 if n = 1 (mod 8) or n = 7 (mod 8)
   5. (2/n) = -1 if n = 3 (mod 8) or n = 5 (mod 8)
      = 0 otherwise

   6. $(a/n) = (n/a)*(-1^{((a-1)(n-1)/4)})$ if a and n are both odd.

The algorithm for the test is really simple. We can pick up a random a and compute (a/n). If n is a prime then (a/n) should be equal to $(a^{((n-1)/2)})\%n$ [as proved by Euler]. If they are not equal then n is composite, and we can stop. Otherwise we can choose more random values for a and repeat the test. We can declare n to be probably prime after some iterations.

Note that we are not interested in calculating Jacobi Symbol (a/n) if n is an even integer because we can trivially see that n isn't prime, except 2 of course.

Let us write a little code to compute Jacobian Symbol (a/n) and for the test:

```
//calculates Jacobian(a/n) n>0 and n is odd
int calculateJacobian(long long a,long long n){
    if(!a) return 0; // (0/n) = 0
    int ans=1;
    long long temp;
    if(a<0){
        a=-a;    // (a/n) = (-a/n)*(-1/n)
        if(n%4==3) ans=-ans; // (-1/n) = -1 if n = 3 ( mod 4 )
    }
    if(a==1) return ans; // (1/n) = 1
    while(a){
        if(a<0){
            a=-a;    // (a/n) = (-a/n)*(-1/n)
            if(n%4==3) ans=-ans;    // (-1/n) = -1 if n = 3 ( mod 4 )
        }
        while(a%2==0){
            a=a/2;    // Property (iii)
            if(n%8==3||n%8==5) ans=-ans;
        }
        swap(a,n);    // Property (iv)
        if(a%4==3 && n%4==3) ans=-ans; // Property (iv)
        a=a%n; // because (a/p) = (a%p / p ) and a%pi = (a%n)%pi if n % pi = 0
        if(a>n/2) a=a-n;
    }
    if(n==1) return ans;
    return 0;
}

/* Iterations determine the accuracy of the test */
bool Solovoy(long long p,int iteration){
    if(p<2) return false;
    if(p!=2 && p%2==0) return false;
    for(int i=0;i<iteration;i++){
        long long a=rand()%(p-1)+1;
        long long jacobian=(p+calculateJacobian(a,p))%p;
        long long mod=modulo(a,(p-1)/2,p);
        if(!jacobian || mod!=jacobian){
            return false;
        }
    }
    return true;
}
```

It is shown that for any composite n, at least half of the a will result in n being declared as composite according to Solovay-Strassen test. This shows that the probability of getting a wrong result after k iterations is $(1/2)^k$. However, it is generally less preferred than Rabin-Miller test in practice because it gives poorer performance.

The various routines provided in the article can be highly optimized just by using bitwise operators instead of them. For example /= 2 can be replaced by ">>= 1", "%2" can be replaced by "&1" and "*= 2" can be replaced by "<<=1". Inline Assembly can also be used to optimize them further.

Practice Problems

Problems involving non-deterministic primality tests are not very suitable for the SRM format. But problems involving modular exponentiation and matrix exponentiation are common. Here are some of the problems where you can apply the methods studied above:

PowerDigit (TCO 06 Online Round 2)
MarbleMachine (SRM 376)
DrivingAround (SRM 342)
PON
PRIC
SOLSTRAS
DIVSUM2 [this one also involves Pollard's Rho Algorithm]

Introduction

Are you familiar with the following situation? You open the Div I Medium and don't know how to approach it, while a lot of people in your room submitted it in less than 10 minutes. Then, after the contest, you find out in the editorial that this problem can be simply reduced to a classical one. If yes, then this tutorial will surely be useful for you.

Problem statement

In this article we'll deal with one optimization problem, which can be informally defined as:

*Assume that we have **N** workers and **N** jobs that should be done. For each pair (worker, job) we know salary that should be paid to worker for him to perform the job. Our goal is to complete all jobs minimizing total inputs, while assigning each worker to exactly one job and vice versa.*

Converting this problem to a formal mathematical definition we can form the following equations:

$\{c_{ij}\}_{N \times N}$ - cost matrix, where c_{ij} - cost of worker i to perform job j.

$\{x_{ij}\}_{N \times N}$ - resulting binary matrix, where $x_{ij} = 1$ if and only if i^{th} worker is assigned to j^{th} job.

$$\sum_{j=1}^{N} x_{ij} = 1, \qquad \forall i \in \overline{1,N}$$

- one worker to one job assignment.

$$\sum_{i=1}^{N} x_{ij} = 1, \qquad \forall j \in \overline{1,N}$$

- one job to one worker assignment.

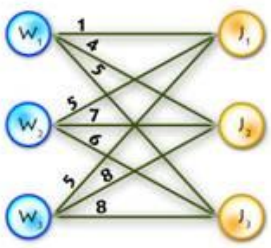
$$\sum_{i=1}^{N} \sum_{j=1}^{N} c_{ij} x_{ij} \to min$$

- total cost function.

We can also rephrase this problem in terms of graph theory. Let's look at the job and workers as if they were a bipartite graph, where each edge between the i^{th} worker and j^{th} job has weight of c_{ij}. Then our task is to find minimum-weight matching in the graph (the matching will consists of **N** edges, because our bipartite graph is complete).

Small example just to make things clearer:

$$\begin{pmatrix} 1 & 4 & 5 \\ 5 & 7 & 6 \\ 5 & 8 & 8 \end{pmatrix} \Leftrightarrow$$



General description of the algorithm

This problem is known as the assignment problem. The assignment problem is a special case of the transportation problem, which in turn is a special case of the min-cost flow problem, so it can be solved using algorithms that solve the more general cases. Also, our problem is a special case of binary integer linear programming problem (which is NP-hard). But, due to the specifics of the problem, there are more efficient algorithms to solve it. We'll handle the assignment problem with the Hungarian algorithm (or Kuhn-Munkres algorithm). I'll illustrate two different implementations of this algorithm, both graph theoretic, one easy and fast to implement with **O(n^4)** complexity, and the other one with **O(n^3)** complexity, but harder to implement.

There are also implementations of Hungarian algorithm that do not use graph theory. Rather, they just operate with cost matrix, making different transformation of it (see [1] for clear explanation). We'll not touch these approaches, because it's less practical for TopCoder needs.

O(n^4) algorithm explanation

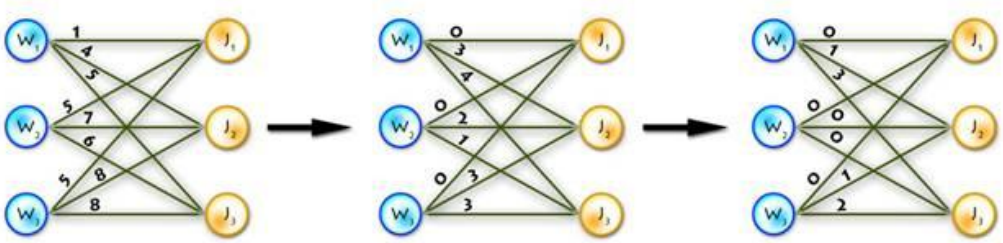
As mentioned above, we are dealing with a bipartite graph. The main idea of the method is the following: consider we've found the perfect matching using only edges of weight 0 (hereinafter called "0-weight edges"). Obviously, these edges will be the solution of the assignment problem. If we can't find perfect matching on the current step, then the Hungarian algorithm changes weights of the available edges in such a way that the new 0-weight edges appear and these changes do not influence the optimal solution.

To clarify, let's look at the step-by-step overview:

Step 0)

A. For each vertex from left part (workers) find the minimal outgoing edge and subtract its weight from all weights connected with this vertex. This will introduce 0-weight edges (at least one).

B. Apply the same procedure for the vertices in the right part (jobs).

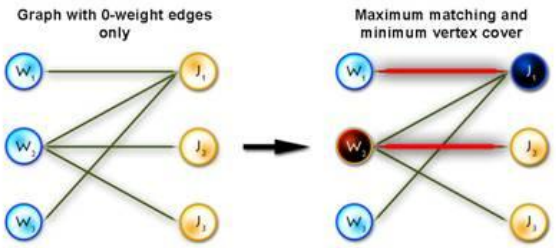


Actually, this step is not necessary, but it decreases the number of main cycle iterations.

Step 1)

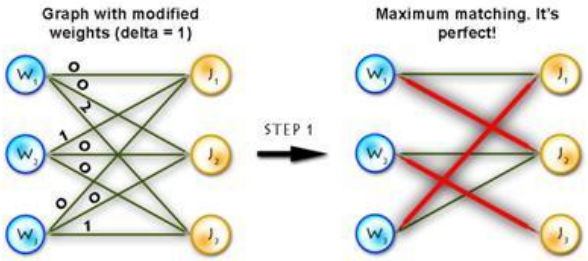
A. Find the maximum matching using only 0-weight edges (for this purpose you can use max-flow algorithm, augmenting path algorithm, etc.).

B. If it is perfect, then the problem is solved. Otherwise find the minimum vertex cover ***V*** (for the subgraph with 0-weight edges only), the best way to do this is to use Köning's graph theorem.



Step 2) Let $\Delta = \min_{i \notin V, j \notin V}(c_{ij})$ and adjust the weights using the following rule:

$$c_{ij} = \begin{cases} c_{ij} - \Delta, i \notin V \wedge j \notin V \\ c_{ij} \quad\quad , i \in V \vee j \in V \\ c_{ij} + \Delta, i \in V \wedge j \in V \end{cases}$$



Step 3) Repeat Step 1 until solved.

But there is a nuance here; finding the maximum matching in step 1 on each iteration will cause the algorithm to become **O(n^5)**. In order to avoid this, on each step we can just modify the matching from the previous step, which only takes **O(n^2)** operations.

It's easy to see that no more than n2 iterations will occur, because every time at least one edge becomes 0-weight. Therefore, the overall complexity is **O(n^4)**.

O(n^3) algorithm explanation

Warning! In this section we will deal with the maximum-weighted matching problem. It's obviously easy to transform minimum problem to the maximum one, just by setting:

$$w(x,y) = -w(x,y), \forall (x,y) \in E$$

or

$$w(x,y) = M - w(x,y), M = \max_{(x,y) \in E} w(x,y)$$

.

Before discussing the algorithm, let's take a look at some of the theoretical ideas. Let's start off by considering we have a complete bipartite graph ***G=(V,E)*** where $V = X \cup Y (X \cap Y = \varnothing)$ and $E \subseteq X \times Y$, ***w(x,y)*** - weight of edge ***(x,y)***.

Vertex and set neighborhood

Let $v \in V$. Then $J_G(v) = \{u \mid (v,u) \in E\}$ is ***v*'s** neighborhood, or all vertices that share an edge with **v**.

Let $S \subseteq V$. Then $J_G(S) = \bigcup_{v \in S} J_G(v)$ is ***S*'s** neighborhood, or all vertices that share an edge with a vertex in **S**.

Vertex labeling

This is simply a function $l : V \rightarrow R$ (for each vertex we assign some number called a label). Let's call this labeling feasible if it satisfies the following condition: $l(x) + l(y) \geq w(x,y), \forall x \in X, \forall y \in Y$. In other words, the sum of the labels of the vertices on both sides of a given edge are greater than or equal to the weight of that edge.

Equality subgraph

Let ***G$_l$=(V,E$_l$)*** be a spanning subgraph of **G** (in other words, it includes all vertices from **G**). If **G** only those edges **(x,y)** which satisfy the following condition: $(x,y) \in E_l \Leftrightarrow (x,y) \in E \wedge l(x) + l(y) = w(x,y)$, then it is an equality subgraph. In other words, it only includes those edges from the bipartite matching which allow the vertices to be perfectly feasible.

Now we're ready for the theorem which provides the connection between equality subgraphs and maximum-weighted matching:

*If **M*** is a perfect matching in the equality subgraph **G$_l$**, then **M*** is a maximum-weighted matching in **G**.*

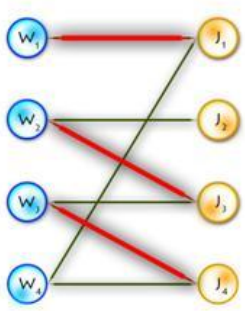
The proof is rather straightforward, but if you want you can do it for practice. Let's continue with a few final definitions:

Alternating path and alternating tree

Consider we have a matching ***M*** **(** $M \subseteq E$ **)**.

Vertex $v \in V$ is called matched if $\exists x \in X : (x,v) \in M \vee \exists y \in Y : (v,y) \in M$, otherwise it is called *exposed (free, unmatched)*.

(In the diagram below, W_1, W_2, W_3, J_1, J_3, J_4 are matched, W_4, J_2 are exposed)



Path ***P*** is called alternating if its edges alternate between **M** and **E\M**. (For example, (W_4, J_4, W_3, J_3, W_2, J_2) and (W_4, J_1, W_1) are alternating paths)

If the first and last vertices in alternating path are exposed, it is called *augmenting* (because we can increment the size of the matching by inverting edges along this path, therefore matching unmatched edges and vice versa). ((W_4, J_4, W_3, J_3, W_2, J_2) - augmenting alternating path)

A tree which has a root in some exposed vertex, and a property that every path starting in the root is alternating, is called an *alternating tree*. (Example on the picture above, with root in W_4)

That's all for the theory, now let's look at the algorithm:

First let's have a look on the scheme of the Hungarian algorithm:

Step 0. Find some initial feasible vertex labeling and some initial matching.

Step 1. If **M** is perfect, then it's optimal, so problem is solved. Otherwise, some exposed $x \in X$ exists; set $S = \{x\}, T = \{\}$. (**x** - is a root of the alternating tree we're going to build). Go to step 2.

Step 2. If $J_{G_l}(S) \neq T$ go to step 3, else $J_{G_l}(S) = T$. Find

$$\Delta = \min_{x \in S, y \in Y \setminus T} (l(x) + l(y) - w(x,y)) \qquad (1)$$

and replace existing labeling with the next one:

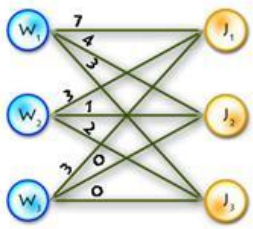
$$l'(v) = \begin{cases} l(v) - \Delta, v \in S \\ l(v) + \Delta, v \in T \\ l(v), otherwise \end{cases} \qquad (2)$$

Now replace G_l with $G_{l'}$

Step 3. Find some vertex $y \in T \setminus J_{G_l}(S)$. If ***y*** is exposed then an alternating path from ***x*** (root of the tree) to ***y*** exists, augment matching along this path and go to step 1. If ***y*** is matched in **M** with some vertex ***z*** add ***(z,y)*** to the alternating tree and set $S = S \cup \{z\}, T = T \cup \{y\}$, go to step 2.

And now let's illustrate these steps by considering an example and writing some code.

As an example we'll use the previous one, but first let's transform it to the maximum-weighted matching problem, using the second method from the two described above. (See Picture 1)



Picture 1

Here are the global variables that will be used in the code:

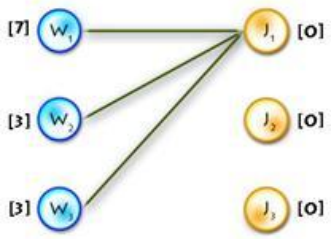
```
#define N 55                //max number of vertices in one part
#define INF 100000000       //just infinity

int cost[N][N];             //cost matrix
int n, max_match;           //n workers and n jobs
int lx[N], ly[N];           //labels of X and Y parts
int xy[N];                  //xy[x] - vertex that is matched with x,
int yx[N];                  //yx[y] - vertex that is matched with y
bool S[N], T[N];            //sets S and T in algorithm
int slack[N];               //as in the algorithm description
int slackx[N];              //slackx[y] such a vertex, that
                            // l(slackx[y]) + l(y) - w(slackx[y],y) = slack[y]
int prev[N];                //array for memorizing alternating paths
```

Step 0:

It's easy to see that next initial labeling will be feasible:

$$l(x) = \max_{(x,y) \in E} (w(x,y))$$
$$l(y) = 0, y \in Y$$

And as an initial matching we'll use an empty one. So we'll get equality subgraph as on Picture 2. The code for initializing is quite easy, but I'll paste it for completeness:



```
void init_labels()
{
    memset(lx, 0, sizeof(lx));
    memset(ly, 0, sizeof(ly));
    for (int x = 0; x < n; x++)
        for (int y = 0; y < n; y++)
            lx[x] = max(lx[x], cost[x][y]);
}
```

The next three steps will be implemented in one function, which will correspond to a single iteration of the algorithm. When the algorithm halts, we will have a perfect matching, that's why we'll have n iterations of the algorithm and therefore **(n+1)** calls of the function.

Step 1

According to this step we need to check whether the matching is already perfect, if the answer is positive we just stop algorithm, otherwise we need to clear ***S***, ***T*** and alternating tree and then find some exposed vertex from the ***X*** part. Also, in this step we are initializing a *slack* array, I'll describe it on the next step.

```
void augment()                          //main function of the algorithm
{
    if (max_match == n) return;         //check wether matching is already perfect
    int x, y, root;                     //just counters and root vertex
    int q[N], wr = 0, rd = 0;           //q - queue for bfs, wr,rd - write and read
                                        //pos in queue
    memset(S, false, sizeof(S));        //init set S
    memset(T, false, sizeof(T));        //init set T
    memset(prev, -1, sizeof(prev));     //init set prev - for the alternating tree
    for (x = 0; x < n; x++)             //finding root of the tree
        if (xy[x] == -1)
        {
            q[wr++] = root = x;
            prev[x] = -2;
            S[x] = true;
            break;
        }

    for (y = 0; y < n; y++)             //initializing slack array
    {
        slack[y] = lx[root] + ly[y] - cost[root][y];
        slackx[y] = root;
    }
```

Step 2

On this step, the alternating tree is completely built for the current labeling, but the augmenting path hasn't been found yet, so we need to improve the labeling. It will add new edges to the equality subgraph, giving an opportunity to expand the alternating tree. This is the main idea of the method; *we are improving the labeling until we find an augmenting path in the equality graph corresponding to the current labeling*. Let's turn back to step 2. There we just change labels using formulas **(1)** and **(2)**, but using them in an obvious manner will cause the algorithm to have **O(n^4)** time. So, in order to avoid this we use a *slack* array initialized in **O(n)** time because we only augment the array created in step 1:

$$slack[y] = \min_{x \in S}(l(x) + l(y) - w(x,y))$$

Then we just need O(n) to calculate a delta Δ (see (1)):

$$\Delta = \min_{y \in Y \setminus T} slack[y]$$

Updating slack:
1) On **step 3**, when vertex ***x*** moves from ***X**S*** to ***S***, this takes **O(n)**.
2) On **step 2**, when updating labeling, it's also takes **O(n)**, because:

$$\Delta = \min_{y \in Y \setminus T} slack[y]$$

So we get **O(n)** instead of **O(n^2)** as in the straightforward approach.
Here's code for the label updating function:

```
void update_labels()
{
    int x, y, delta = INF;             //init delta as infinity
    for (y = 0; y < n; y++)            //calculate delta using slack
        if (!T[y])
            delta = min(delta, slack[y]);
    for (x = 0; x < n; x++)            //update X labels
        if (S[x]) lx[x] -= delta;
    for (y = 0; y < n; y++)            //update Y labels
        if (T[y]) ly[y] += delta;
    for (y = 0; y < n; y++)            //update slack array
        if (!T[y])
            slack[y] -= delta;
}
```

Step 3

In step 3, first we build an alternating tree starting from some exposed vertex, chosen at the beginning of each iteration. We will do this using breadth-first search algorithm. If on some step we meet an exposed vertex from the ***Y*** part, then finally we can augment our path, finishing up with a call to the main function of the algorithm. So the code will be the following:

1) Here's the function that adds new edges to the alternating tree:

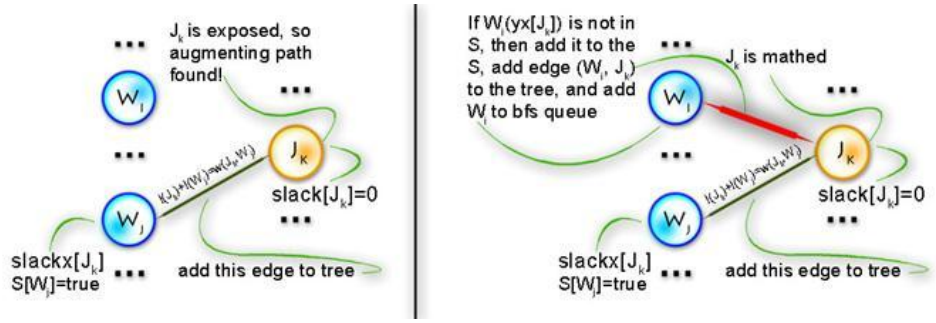
```
void add_to_tree(int x, int prevx)
//x - current vertex,prevx - vertex from X before x in the alternating path,
//so we add edges (prevx, xy[x]), (xy[x], x)
{
    S[x] = true;                    //add x to S
    prev[x] = prevx;                //we need this when augmenting
    for (int y = 0; y < n; y++)    //update slacks, because we add new vertex to S
        if (lx[x] + ly[y] - cost[x][y] < slack[y])
        {
            slack[y] = lx[x] + ly[y] - cost[x][y];
            slackx[y] = x;
        }
}
```

3) And now, the end of the ***augment()*** function:

```
//second part of augment() function
    while (true)                    //main cycle
    {
        while (rd < wr)             //building tree with bfs cycle
        {
            x = q[rd++];                //current vertex from X part
            for (y = 0; y < n; y++)         //iterate through all edges in equality graph
                if (cost[x][y] == lx[x] + ly[y] &&  !T[y])
                {
                    if (yx[y] == -1) break;   //an exposed vertex in Y found, so
                                              //augmenting path exists!
                    T[y] = true;        //else just add y to T,
                    q[wr++] = yx[y]; //add vertex yx[y], which is matched
               //with y, to the queue
                    add_to_tree(yx[y], x);  //add edges (x,y) and (y,yx[y]) to the tree
                }
            if (y < n) break;           //augmenting path found!
        }
        if (y < n) break;               //augmenting path found!

        update_labels();            //augmenting path not found, so improve labeling
        wr = rd = 0;
        for (y = 0; y < n; y++)
        //in this cycle we add edges that were added to the equality graph as a
        //result of improving the labeling, we add edge (slackx[y], y) to the tree if
        //and only if !T[y] &&  slack[y] == 0, also with this edge we add another one
        //(y, yx[y]) or augment the matching, if y was exposed
```

```
            if (!T[y] &&  slack[y] == 0)
            {
                if (yx[y] == -1)      //exposed vertex in Y found - augmenting path exists!
                {
                    x = slackx[y];
                    break;
                }
                else
                {
                    T[y] = true;           //else just add y to T,
                    if (!S[yx[y]])
                    {
                        q[wr++] = yx[y]; //add vertex yx[y], which is matched with
                                         //y, to the queue
                        add_to_tree(yx[y], slackx[y]); //and add edges (x,y) and (y,
                                                       //yx[y]) to the tree
                    }
                }
            }
        if (y < n) break;               //augmenting path found!
    }
    if (y < n)            //we found augmenting path!
    {
        max_match++;  //increment matching
        //in this cycle we inverse edges along augmenting path
        for (int cx = x, cy = y, ty; cx != -2; cx = prev[cx], cy = ty)
        {
            ty = xy[cx];
            yx[cy] = cx;
            xy[cx] = cy;
        }
        augment();     //recall function, go to step 1 of the algorithm
    }
}//end of augment() function
```

The only thing in code that hasn't been explained yet is the procedure that goes after labels are updated. Say we've updated labels and now we need to complete our alternating tree; to do this and to keep algorithm in $O(n^3)$ time (it's only possible if we use each edge no more than one time per iteration) we need to know what edges should be added without iterating through all of them, and the answer for this question is to use BFS to add edges only from those vertices in **Y**, that are not in **T** and for which **slack[y] = 0** (it's easy to prove that in such way we'll add all edges and keep algorithm to be $O(n^3)$). See picture below for explanation:

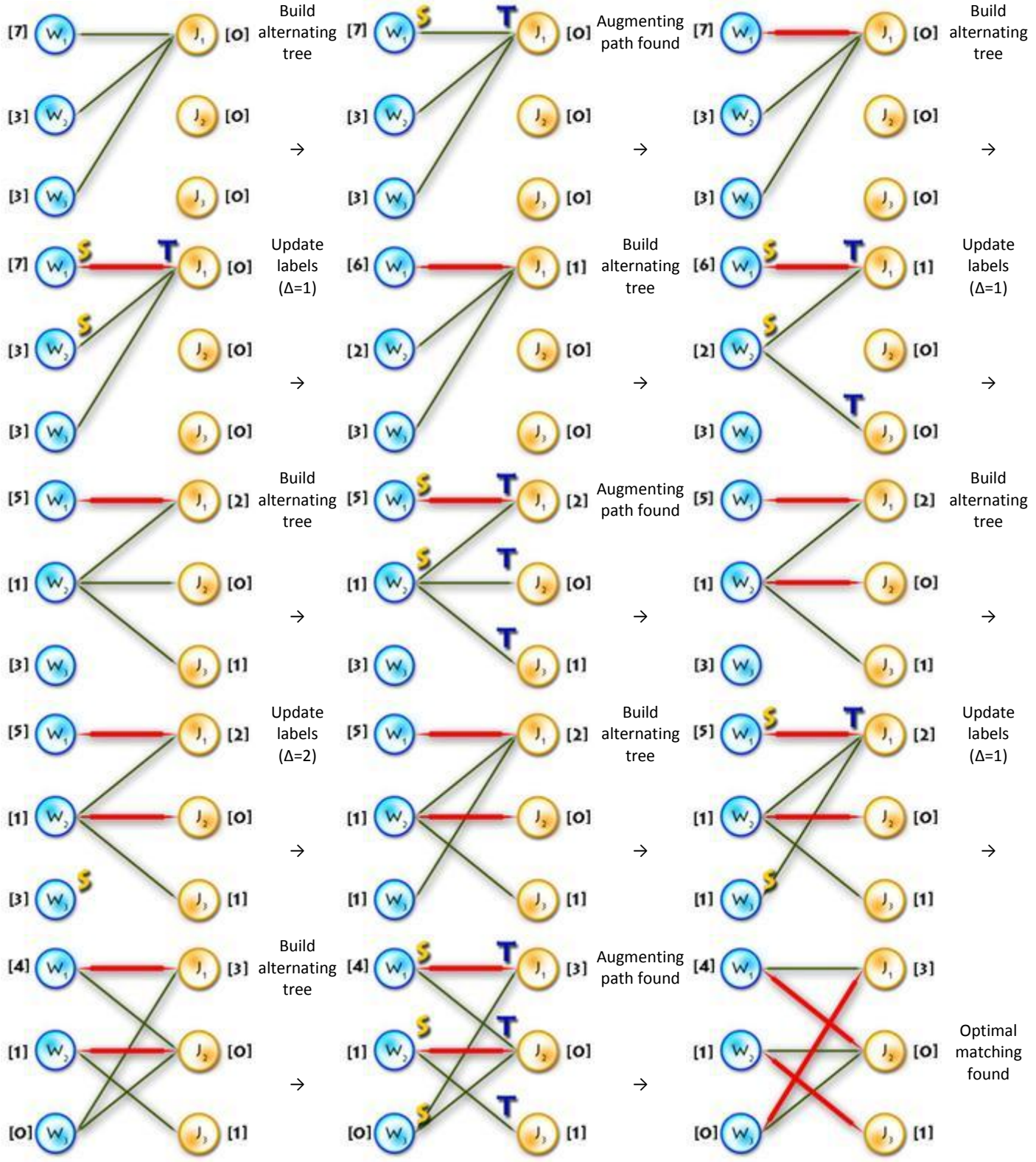


At last, here's the function that implements Hungarian algorithm:

```
int hungarian()
{
    int ret = 0;                    //weight of the optimal matching
    max_match = 0;                  //number of vertices in current matching
    memset(xy, -1, sizeof(xy));
    memset(yx, -1, sizeof(yx));
    init_labels();                  //step 0
    augment();                      //steps 1-3
    for (int x = 0; x < n; x++)     //forming answer there
        ret += cost[x][xy[x]];
    return ret;
```

}

To see all this in practice let's complete the example started on step 0.



Finally, let's talk about the complexity of this algorithm. On each iteration we increment matching so we have **n** iterations. On each iterations each edge of the graph is used no more than one time when finding augmenting path, so we've got $\mathbf{O(n^2)}$ operations. Concerning labeling we update *slack* array each time when we insert vertex from ***X*** into ***S***, so this happens no more than n times per iteration, updating *slack* takes **O(n)** operations, so again we've got $\mathbf{O(n^2)}$. Updating labels happens no more than n time per iterations (because we add at least one vertex from ***Y*** to ***T*** per iteration), it takes **O(n)** operations - again $\mathbf{O(n^2)}$. So total complexity of this implementation is $\mathbf{O(n^3)}$.

Some practice

For practice let's consider the medium problem from SRM 371 (div. 1). It's obvious we need to find the maximum-weighted matching in graph, where the ***X*** part is our players, the ***Y*** part is the opposing club players, and the weight of each edge is:

$$w(x,y) = \begin{cases} 0, us[x] < them[y] \\ 1, us[x] = them[y] \\ 2, us[x] > them[y] \end{cases}$$

Though this problem has a much simpler solution, this one is obvious and fast coding can bring more points.

Try this one for more practice. I hope this article has increased the wealth of your knowledge in classical algorithms… Good luck and have fun!