GB21802 - Programming Challenges

Week 1 - Data Structures

Claus Aranha
caranha@cs.tsukuba.ac.jp

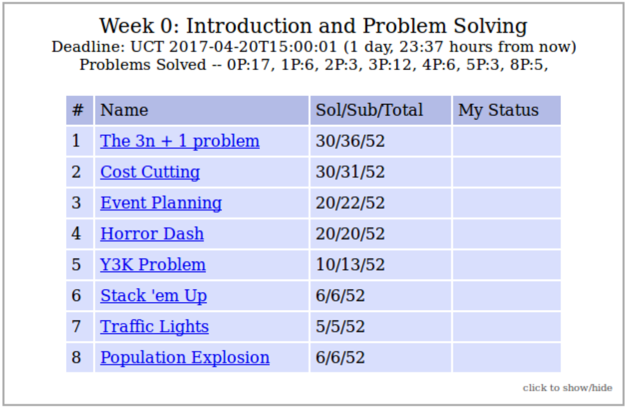
College of Information Science

2017-04-22,25

Last updated April 19, 2017

Results for the Previous Week

Here are the results for last week:



Week 0: Introduction and Problem Solving
Deadline: UCT 2017-04-20T15:00:01 (1 day, 23:37 hours from now)
Problems Solved -- 0P:17, 1P:6, 2P:3, 3P:12, 4P:6, 5P:3, 8P:5,

#	Name	Sol/Sub/Total	My Status
1	The 3n + 1 problem	30/36/52	
2	Cost Cutting	30/31/52	
3	Event Planning	20/22/52	
4	Horror Dash	20/20/52	
5	Y3K Problem	10/13/52	
6	Stack 'em Up	6/6/52	
7	Traffic Lights	5/5/52	
8	Population Explosion	6/6/52	

click to show/hide

Hope you enjoyed the warm up!

Comments from e-mails and questions – 1

Submission with Java

Two students had "runtime error" with Java last week – don't forget that your start class MUST be called **Main**.

```
class Main {
  public static void main(String[] args) {

     // do something...

   }
}
```

Comments from e-mails and questions – 2

Input/Output

Two other students had problems because their program printed “Please enter a number”.

You are very kind, but please **follow the specifications** strictly!

Format for MANABA submission

One student asked if the code for MANABA had to be the same as the code for UVA.

Yes. The code you submit on MANABA must be **exactly the same** as the code you submitted for UVA.

Short comments about the problems:

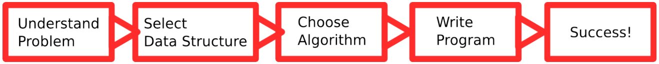
- Cost Cutting, Event Planning, Horror Dash – Easiest problems (find mean, find min, find min);
- 3n+1 – Still easy, but a few traps – example of **Memoization**;
- Y3K Problem – Still easy, but skip year can be a bit troublesome.

Data Structures
CP Book Chapter 2

Motivation: Why study data structures?

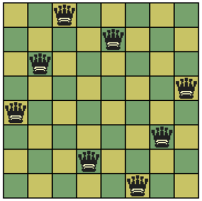
Correct Data Structure makes the problem Easier

- The program becomes simpler;
- Less Bugs;
- Solution becomes faster;
- Other good things



In this class, we will focus on **implementation.** See the “Data Structures” class for the theory.

Example 1: 8 Queen Problem (UVA 750)

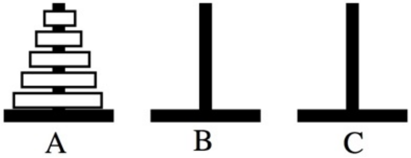


How to represent a solution?

- Queen i at position $x, y \rightarrow (64!/56!)$ total solutions
- Queen i at column $c \rightarrow 8^8$ total solutions
- Permutation of rows $\rightarrow 8!$ total solutions

Image by Lee Daniel Crocker. CC-BY-SA 3.0

Example 2: The Towers of Hanoi



- You have N disks and K poles. Each disk has unique size s_i.
- A disk i can be moved from one pole to another.
- A move of disk i to pole k is only valid if k has no disks smaller than i
- Find the list of moves to move all disks from pole 1 to pole K.

How do you represent the data in this problem?

Another way to visualize the Towers of Hanoi

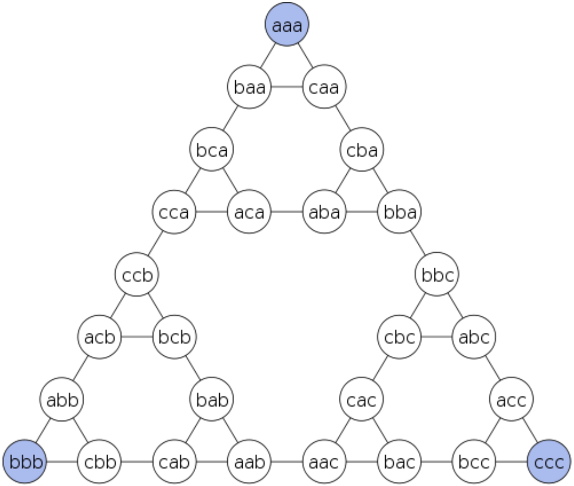
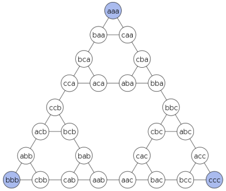


Image created by nonenmac

Explaining the Tower of Hanoi Data Structure

- Each node identifies one state in the problem;
- The string represents the position of each disk;
- At most **Three** state transitions at any time;
- We can solve the problem from **any start state** to **any end state**;
- Just find a path between the states!
- (just beware of state explosion)



One Important Reminder:

Know your libraries!

Array, Tree, Vector, Matrix, Graph... How do you implement them?

Most of these have standard functions in the library (STL, java.util), but do you remember it?

Use it many times until you don’t need to google anymore.

Complex data structures (trees, edge matrices) are great candidates for your personal library!

The simple array!

Arrays are the simplest data structure, but also the most often used.

- Preserve *Programmer Efficiency*
- No worries about pointers;
- Random access;
- Library Functions;

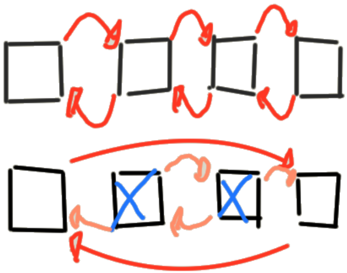
Make arrays, not lists

Example: Army Buddies (12356)

A line of soldiers, each soldier must know who is in his **left** and **right** neighbor.

If some soldier dies, you need to **update** the left and right neighbors.

Linked List Approach: Regular idea.



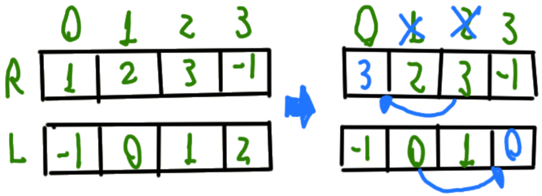
Make arrays, not lists

Example: Army Buddies (12356)

A line of soldiers, each soldier must know who is in his **left** and **right** neighbor.

If some soldier dies, you need to **update** the left and right neighbors.

Array Approach: $L[R[r]] = L[l]; R[L[l]] = R[r];$



Implementing arrays

```
#include <vector>             // for C++ vectors

int arr[5] = {7,7,7};         // arr = {7,7,7,0,0}
vector<int> v(5, 5);          // v = {5,5,5,5,5}

int x = arr[2] + v[2];        // x = 12

arr[5] = 5;                   // Runtime error
cout << v[7];                 // 0 !! Be careful.

v.push_back(6);               // v = {5,5,5,5,5,6}
```

Learn the different functions of your language of choice!

Implementation Matters: Resetting

```
#include <vector>
#include <string.h>

vector<int> v(10000,7)

memset(v, 0, 10000*__SIZEOF_INT__);          // Method 1
fill(v.begin(), v.end(), 0);                 // Method 2
for (int i = 0; i < 10000; i++) v[i] = 0;   // Method 3
v.assign(v.size(), 0);                       // Method 4
```

Method	executable size		Time Taken (in sec)	
	-O0	-O3	-O0	-O3
1. memset	17 kB	8.6 kB	0.125	0.124
2. fill	19 kB	8.6 kB	13.4	0.124
3. manual	19 kB	8.6 kB	14.5	0.124
4. assign	24 kB	9.0 kB	1.9	0.591

Queue and Stacks

Queues and Stacks are useful to simplify common cases of vectors

Queue example: List of nodes to visit in pathfinding

```
#include <queue>
#include <vector>
#include <utility>

// index and neighbor list
queue <pair <int, vector <int>>> visit_list;

// ... data initialization

pair <int, vector <int>> cur = visit_list.front();

for (int i = 0; i < neighbor[cur.first]; i++)
   visit_list.push(cur.second[i]);
```

Queue and Stacks

Queues and Stacks are useful to simplify common cases of vectors

Stack Example: Testing if a set of parenthesis is balanced.

```
#include <stack>
stack<char> s;
char c;

while(cin >> c) {
  if (c == '(') s.push(c);
  else {
    if (s.size() == 0) { s.push('*'); break; }
    s.pop();
  }
}
cout << (s.size() == 0 ? "balanced" : "unbalanced");
```

Sorting

Sorting is an extremely important operation. Many algorithms begin or end with sorting the data.

- Finding the n-th highest value;
- Finding duplicate values;
- Pre-processing data for binary search;
- etc...

You should know how to sort in your chosen language with your eyes closed!

Sorting Example – Vito's Family (UVA 10041)

Problem description

A gangster is looking for a new house. The distance from the new house to **each and all** family members should be minimal.

This program can be summarized as "find the median of the addresses and calculate the distance to all the houses."

```
#include <algorithm>

int addr[m] // positions (0-5000) of each house.
// read positions from input.

sort(addr,addr+m);
// if vector, sort(addr.begin(), addr.end())
med = add[(m/2)];
```

Sorting with specific sorting function

Remember, in the function, the main comparison should come first.

```
#include <algorithm>
#include <vector>
#include <string>
struct team{ string n; int p; int g;
             team(string _n, int _p, int _g) :
                n(_n), p(_p), g(_g){} };

bool cmp(team a, team b) {
  if (a.p != b.p) return a.p < b.p;
  if (a.g != b.g) return a.g > b.g;
  return 1; }

vector<team> v;
sort(v.begin(), v.end(), cmp); // sort using cmp
reverse(v.begin(), v.end()); // and reverse
```

To sort or not to sort?

Let’s say we want to find an specific value k

Checking each value in the array with n elements

- Cost for searching once: $O(n)$
- Cost for searching m times **in the same array**: $O(m * n)$

Sort and binary search

- Cost for searching once: $O(n\log n + \log n)$
- Cost for searching m times in the same array: $O(n\log n + m\log n)$

The best one depends on the problem!

Binary Search

You can use algorithm::lower_bound and algorithm::upper_bound

```
#include <iostream>
#include <algorithm>
#include <vector>
int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  vector<int> v(myints,myints+8);
  sort (v.begin(), v.end());

  vector<int>::iterator low,up;
  low= lower_bound (v.begin(), v.end(), 20);
  up = upper_bound (v.begin(), v.end(), 20);

  cout <<"lower at "<<(low-v.begin())<< '\n';
  cout <<"upper at "<<(up -v.begin())<< '\n';

  return 0; // up and low are memory indexes. }
```

Maps and Sets

Maps and sets are useful if you have a group of unique items. It is easy to test for the **Presence of Abscense** of a particular item.

Also, if the number of elements is too large (example, problem CD), maps are usually really fast.

Map implementation (1)

```
#include <map>
#include <set>

set<int> used_values;          used_values.clear();
map<string, int> mapper;       mapper.clear();

mapper["john"] = 78;    used_values.insert(78);
mapper["billy"] = 69;   used_values.insert(69);
mapper["andy"] = 80;    used_values.insert(80);
mapper["steven"] = 77; used_values.insert(77);
mapper["felix"] = 82;   used_values.insert(82);

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

Map implementation (2)

```
// interesting usage of lower_bound and upper_bound
// display data between ["f".."m") ('felix', 'john')
for (map<string, int>::iterator it =
     mapper.lower_bound("f");
     it != mapper.upper_bound("m"); it++)
  printf("%s %d\n",
         ((string)it->first).c_str(),
         it->second);

// O(log n) search of a value:
set<int>::iterator f = used_values.find(79);
if (f == used_values.end())
  cout << "not found!\n";
else
  cout << *f;
```

Union-Find Disjoint Set

Sometimes we need to create our own data sets.

- The UFDS represent a **set of items** that can be in one of many **groups**
- For example, we can have a set of **people** who can be in one of many **families**
- This can be useful to finding **unconnected components** in graphs
- A good implementation of UFDS can test if a item belongs to a group in close to O(1);

UFDS Implementation (1/3)

```
typedef vector<int> vi; // use shorthand for vectors

class UnionFind {
private:
  vi p, rank, setSize;  // vi -- vector <int>
  int numSets;
public:
  UnionFind(int N) {
    setSize.assign(N, 1); // each element in a set
    numSets = N;
    rank.assign(N, 0);    // everyone is root
    p.assign(N, 0);
    for (int i = 0; i < N; i++) p[i] = i; }

  int findSet(int i) {
    return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
```

UFDS Implementation (2/3)

```
  // continue UnionFind class

  bool isSameSet(int i, int j)
     { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) { numSets--;
    int x = findSet(i), y = findSet(j);
    // rank is used to keep the tree short
    if (rank[x] > rank[y])
      { p[y] = x; setSize[x] += setSize[y]; }
    else
      { p[x] = y; setSize[y] += setSize[x];
        if (rank[x] == rank[y]) rank[y]++; } } }
  int numDisjointSets() { return numSets; }
  int sizeOfSet(int i) { return setSize[findSet(i)]; }
};
```

UFDS Implementation (3/3)

```
int main() {
  UnionFind UF(5); // create 5 disjoint sets
  printf("%d\n", UF.numDisjointSets()); // 5
  UF.unionSet(0, 1);
  printf("%d\n", UF.numDisjointSets()); // 4
  UF.unionSet(2, 3);
  printf("%d\n", UF.numDisjointSets()); // 3
  UF.unionSet(4, 3);
  printf("%d\n", UF.numDisjointSets()); // 2
  printf("isSameSet(0, 3) = %d\n", UF.isSameSet(0, 3));
    // will return 0 (false)
  printf("isSameSet(4, 3) = %d\n", UF.isSameSet(4, 3));
    // will return 1 (true)
  UF.unionSet(0, 3);
  printf("%d\n", UF.numDisjointSets()); // 1
}
```

Using Bitmasks

A bitmask is a lightweight version of a bitset. You can use an unsigned integer or an unsigned long directly as a proxy for a bitset.

```
#include <iostream>

using namespace std;

int main() {
    unsigned long bb = 3432;
    unsigned long kk = 14;
    cout << (bb & kk) << endl;
    cout << (bb << 3) << endl;
}
```

In a programming contest, they are very useful for quickly manipulating sets (specially if you need to modify a large number of members at the same time).

When to use bitmasks

In programming challenges, bitmasks are useful for quickly manipulating sets of items. Specially if you need to modify multiple sets at the same time.

$S = 34 = 100010$

Can be seen as a set with elements 1 and 5 present.
The index increase with the digit significance.

In regular programming, bitmasks are often used to quickly set/check flags or options.

Bitmasks in regular programming

- Parameter setting/testing

```
Gdx.gl.glClear( GL20.GL_COLOR_BUFFER_BIT |
                GL20.GL_DEPTH_BUFFER_BIT );
```

- Collision/Filtering in computer graphics

```
if( isFilled(sprite1Pixel) && isFilled(sprite2Pixel))
    return true;
```

Binary Operatons on Bitmasks (2)

- Multiply/Divide an integer by two :: shift bits left, right

```
S               = 34          =  100010
S = S << 1 = S*2 = 68 = 1000100
S = S >> 2 = S/4 = 17 =   10001
S = S >> 1 = S/2 =  8 =    1000
```

- To check if the ith item is on the set, use bitwise AND operation, (T = S & (1 « j)) and test if the result is not zero.

```
S               = 34          =  100010
j = 3, 1 << j                 =  001000
i = 1, 1 << 1                 =  000010
                                 ------
Tj= S & ( 1 << j)             =  000000   = 0 # 3 is not set
Ti= S & ( 1 << i)             =  000010 != 0 # 1 is set
```

Binary Operations on Bitmasks (2)

- To set/turn on the jth item, use bitwise OR operation S |= (1 « j)

```
S              = 34        =  100010
j = 3, 1 << j              =  001000
                              ------ OR (S |= 1 << j)
S              = 42        =  101010
```

- To set/turn off the jth item, use bitwise AND operation S &= (1 « j)

```
S              = 50        =  110010
j = (1<<5)|(1<<3)          =  101000 # unset items 5,3
~j                         =  010111
                              ------
S &= ~(j)                  =  010010 # 18
```

Have some code with the previous examples!

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

int main() {
    unsigned int S = 34;
    cout << (S<<1) << endl;
    cout << ((S<<1)>>2) << endl;
    cout << (((S<<1)>>2)>>1) << endl << endl;
    cout << (S & (1 << 3)) << endl;
    cout << (S & (1 << 1)) << endl << endl;
    cout << (S | (1 << 3)) << endl;
    S = 50;
    cout << (S & ~((1 << 5)|(1<<3))) << endl;
}
```

Bitsets

Bitsets can also be used in place of integer bitmasks:

```
#include <bitset>
#include <iostream>

using namespace std;

int main() {
    bitset<12> b(3432);
    cout << "3432 in binary is " << b << endl;

    bitset<12> k(14);
    cout << (b & k) << endl;
    cout << (b<<3) << endl;
}
```

Bitsets and Bitmasks

Most bitmask operations are also supported by bitsets. (But you can't mix them!)

- All binary operations are supported;
- Bitsets are of fixed size, but not limited to 32 and 64 bits.
- Bitset standard output is a bit string, not a decimal (but outputing a decimal is possible);
- But maybe bitmasks are more efficient?

More information about bitmasks

`http://www.drdobbs.com/`

`the-standard-librarian-bitsets-and-bit-v/184401382`

And many more examples

These are just a few examples of data structures that are common in programming challenges. There are many more that you should study and add to your library. For example:

- Binary Tree (2.3)
- Segment Tree (2.4.3)
- Hashing (2.3)
- etc...

Hints

Jolly Jumpers

Test a “jolly” sequence.

- Very easy problem, convert the input correctly?
- Do you need to store the input values?
- What information do you need to memorize?

Newspaper

This is an example of a **DAT** – Direct Address Table – a kind of hashing.

Be careful that any symbol can appear in the text – negative values from signed chars are a common source of bugs.

Army Buddies

We need to keep track of changes in the neighbors of the soldiers.

The time limit is very tight.

You cannot afford to use a O(n) method to update the army.

One idea is the two-array structure discussed earlier.

Grid Successors

In this exercise, you need to find a loop in the sequence of grids.

What is the easiest structure to store this loop?

Football (aka Soccer)

The problem is easy, but you need to know the library to deal with the complex input.

File Fragmentation

Find the original bit string.

- Understanding the problem: What are looking for here?
- Finding out the correct combination:
    - Method1: Elimination;
    - Method2: Counting;

CD

The problem is very easy: given two collections of numbers, find the numbers in both collections.

However, the input data is HUGE – you need to use an efficient data structure.

War

You need to use the UFDS as a basis, but you need to add **something extra.** This one is a challenge for you!

Some Extra Stuff...

How many sorting Algorithms are there?

- bubblesort
- insertion sort
- selection sort
- heapsort
- mergesort
- quicksort
- radix sort
- bin sort
- gnome sort
- library sort
- comb sort
- tree transversal
- sorting networks
- cocktail shaking sort
- bucket sort
- bogo sort
- bitonick sort
- ...
- And many more!

How many sorting Algorithms are there?

- bubblesort
- insertion sort
- selection sort
- heapsort
- mergesort
- quicksort
- radix sort
- bin sort
- gnome sort
- library sort
- comb sort
- tree transversal
- sorting networks
- cocktail shaking sort
- bucket sort
- bogo sort
- bitonick sort
- ...
- And many more!

(╯°□°)╯︵ ┻━┻

What do sorting algorithms sound like?

sorting.mp4