# GB21802 - Programming Challenges

## Week 1 - Data Structures

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Science

2018-04-27,5-7

Last updated May 7, 2018

# Results for the Previous Week

Here are the results for last week:

## Week 0: Introduction and Problem Solving

Deadline: 2018/4/26 23:59:59 (expired)

Problems Solved -- 0P:3, 1P:1, 2P:3, 3P:12, 4P:5, 6P:2, 7P:1, 8P:16,

| # | Name | Sol/Sub/Total | My Status |
|---|---|---|---|
| 1 | The 3n + 1 problem | 36/38/43 | |
| 2 | Cost Cutting | 39/39/43 | |
| 3 | Event Planning | 27/29/43 | |
| 4 | Horror Dash | 33/33/43 | |
| 5 | Y3K Problem | 21/23/43 | |
| 6 | Stack 'em Up | 18/19/43 | |
| 7 | Traffic Lights | 18/18/43 | |
| 8 | Population Explosion | 18/18/43 | |

Hope you enjoyed the warm up!

# More about input...

Do not enter the input by hand! Create an input file, and run your program with the input file to save time and get more precise results.

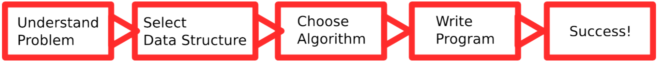
```
$ cat - > input.in
1 10
10 1
10 10
$ g++ 100.cpp
$ ./a.out < input.in
1 10 20
10 1 20
10 10 7
```

# Data Structures
## CP Book Chapter 2

# Motivation: Why study data structures?

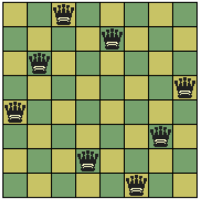
**Correct Data Structure makes the problem Easier**

- 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 0: 8 Queen Problem (UVA 750)

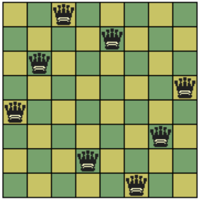


For a board of size $n \times n$, find how many safe configurations of $n$ queens exist.

Because we need to find how many configurations exist, we need to test “all” configurations.

```
for i = 0 to #configurations do
  sum = testIfConfigurationIsSafe(i)
```

# Example 0: 8 Queen Problem (UVA 750)



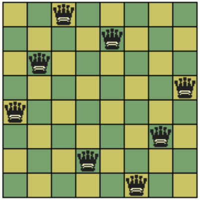
How can we represent a configuration?

Idea one: Represent the position $x, y$ of every queen:

```
Vector conf<array[n*2]>;
conf[0] = {0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0}
conf[1] = {0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,1}
conf[2] = {0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,0, 0,2}
```

Total configurations: $n^{n^2}$

# Example 0: 8 Queen Problem (UVA 750)



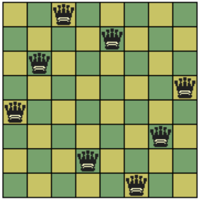
How can we represent a configuration?

Idea one: Represent the row *r* of every queen:

```
Vector conf<array[n]>;
conf[0] = {0,0,0,0,0,0,0,0}
conf[1] = {0,0,0,0,0,0,0,1}
conf[2] = {0,0,0,0,0,0,0,2}
```

Total configurations: $n^n$

# Example 0: 8 Queen Problem (UVA 750)



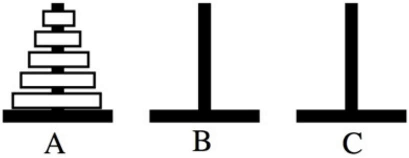
How can we represent a configuration?

Idea one: Represent a permutation of row positions.

```
Vector conf<array[n]>;
conf[0] = {0,1,2,3,4,5,6,7}
conf[1] = {0,1,2,3,4,5,7,6}
conf[2] = {0,1,2,3,4,6,5,7}
```

Total configurations: $n!$ or $(n \times n-1 \times n-2 \times n-3 \ldots)$

# Example 1: 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

A string with “n” disks, from smaller to larger.

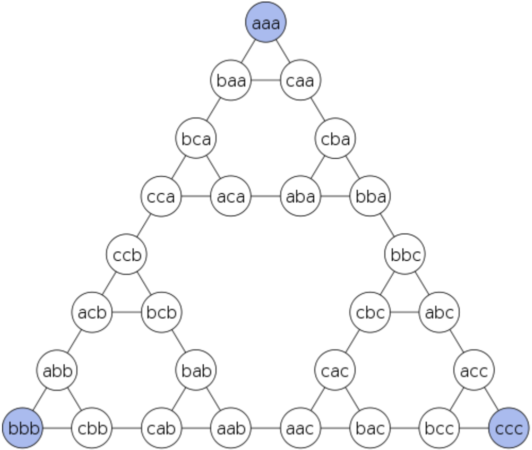



Image created by nonenmac

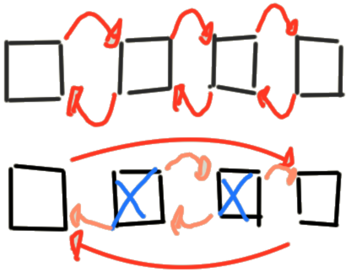
# Example 2: Army Buddies (UVA 12356)
## Problem Description

- There is a line of $S$ soldiers: $0, 1, 2, 3, 4, 5, 6, 7$
- There are $B$ bomb attacks that kill all soldiers from $i$ to $j$:
  - 2,4
  - 6,7
  - 1,1
- After each bomb attack, list the surviving soldier to the left and to the right.
  - 1,5
  - 5,*
  - *,5

How do we solve this problem?

# Example 2: Army Buddies (UVA 12356)

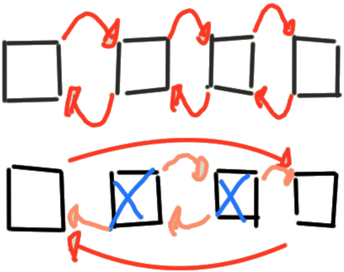
## A solution using linked lists



- Represente the line of soldiers as a linked list.
- Find the first soldier ($O(n)$)
- Find the second soldier ($O(n)$)
- Print the neighbors, and update the list.

# Example 2: Army Buddies (UVA 12356)

## A solution using linked lists



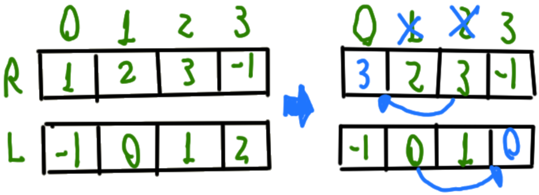
What is the problem with this solution?

- $1 \leq S \leq 10^5$
- $1 \leq B \leq 10^5$

Can you think of a different solution? (10 minutes)

# Example 2: Army Buddies (UVA 12356)

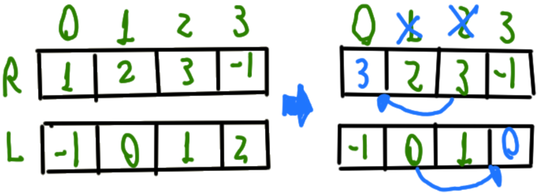
## A solution using arrays




- Problem: We do not want to update ALL soldiers, just the edge soldiers.
- Idea: Neighbor Array
  - Array of Right side neighbors (R)
  - Array of Left side neighbors (L)
- Question: how do we update R and L after a bomb $(r, l)$ explodes?

# Example 2: Army Buddies (UVA 12356)

## A solution using arrays




- Problem: We do not want to update ALL soldiers, just the edge soldiers.
- Idea: Neighbor Array
  - Array of Right side neighbors (R)
  - Array of Left side neighbors (L)
- Question: how do we update R and L after a bomb $(r, l)$ explodes?

  $L[R[r]] = L[l]; R[L[l]] = R[r];$

# Thinking about Data Structures

- Choice of Data structures can change the efficiency of a program;
- Choice of Data Structures can also change the Implementation Complexity of a solution;
- Efficiency $\times$ Complexity Balance:
  - Can you use Data structures from the standard library?
  - Do you need special modifications for the solution?
- Every solution begins with the choice of data structure! Sometimes it is obvious, sometimes not...

# The simple array!

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

## Merits

- Easy to implement! No worries about pointers;
- Can simulate pointers using index operations;
- Many library Functions;

## Concerns

- Reordering many items can be expensive;

# Implementing arrays/vectors (C++)

```
#include <vector>

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}
```

Trying to access indexes outside of an array is a common source of Runtime Errors (RTE)

# How do you reset an array?

## Implementation matters

```
#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    |
```

# Sorting and Searching in Arrays and Vectors

**Consider this problem – Vito's Family (UVA 10041)**

**Input:** You receive a list of street addresses:

10, 20, 10, 10, 40, 80, 30, 90, 20, 55, 20

**Output:** You have to choose the address that *minimizes* the distance to all other addresses:

10: $0 + 10 + 0 + 0 + 30 + 70 + 20 + 80 + 10 + 45 + 10 = 275$
40: $30 + 20 + 30 + 30 + 0 + 40 + 10 + 50 + 20 + 15 + 20 = 265$

How do we solve this problem?

# Sorting and Searching in Arrays and Vectors

- The solution to this problem is the Median house number.
- To find the median, we sort the address array, and select the middle index.

```
#include <algorithm>

int addr[11] = {10, 20, 10, 10, 40,
                80, 30, 90, 20, 55, 20};

sort(addr,addr+11);

result = add[5];
```

# Using Sorting

Sorting can be used for many, many things:

- Finding the Highest *n* values
- Finding duplicate Values
- Binary Search
- Pre-processing for many algorithms

# Binary Search in an vector

algorithm::lower_bound and algorithm::upper_bound will find the indexes for the value you want to search.

```
#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.
```

# Sorting with specific sorting function

Imagine you need to sort by number of points (bigger is best), penalty (smaller is best), and name (alphabetical order)

```
#include <algorithm>
#include <vector>
#include <string>
struct team{ string name; int point; int penal;
             team(string _n, int _po, int _pe) :
               name(_n), point(_p), penal(_g){} };

bool cmp(team a, team b) {
  if (a.point != b.point) return a.point > b.point;
  if (a.penal != b.penal) return a.penal < b.penal;
  return strcmp(a.name,b.name); }

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

# Bitmasks

## A small sidequest...

Bitmasks are lightweight sets of booleans.

We can use integers or long integers to represent a set of booleans, and bitwise operations to manipulate this set.

There are many uses for bitmasks in Programming Challenges:

- You can use them as indexes of sets. Ex: Partial distances in full graph paths;
- You can use them to quickly operate on true/false values;
- You can use them to manipulate states in simulations;
- etc...

# 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;
}
```

You can also use the bitset class, which supports all of these operations, but can't really be used for indexing arrays.

# 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");
```

# Problem Example: CD – 11849

**Input:**

- Jack CD collection: Up to $10^6$ CDs, with ID up to $10^9$
- Jill CD collection: Up to $10^6$ CDs, with ID up to $10^9$

**Output:**

- How Many CDs are in both Collections?

# Problem Example: CD – 11849

Naive Solution:

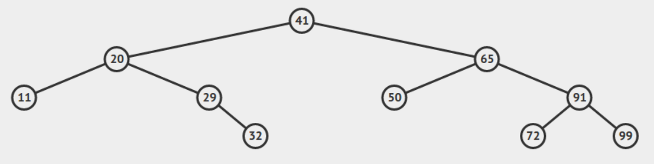
1. Store all IDs in collection 1 in a Vector (n)
2. Sort the Vector (nlogn)
3. For each ID in collection 2, test if it exists in Vector with Binary Search (nlogn)

Total Cost: $n + n\log n + n\log n$

How can we improve?

- Using a Balanced Search Tree – $n\log n + n\log n$
- Using a Hash Table – $nH + nH$

# Balanced Search Trees




- *Search Trees* Keep items in an ordered relationship.
- For example: Left children always have smaller values, Right children always have larger values;
- Insertion/Search/Deletion in a tree costs $O(h)$, where $h$ is the height of the tree;
- For a tree with $n$ elements, the minimum height is $\log n$
- For a balanced tree, the maximum height is also $\log n$
- How to keep the tree balanced?

# Balanced Search Trees

## How to keep the tree balanced?

There are many Tree implementations/algorithms for keeping an BST balanced, and minimizing the tree height efficiently:

- AVL Tree (Adelson-Velskii-Landis);
- Red-Black Tree;
- B-Tree;
- Splay Tree;

However, in a programming context (or even day to day life), implementing these trees from scratch is Dangerous.

Luckly, most standard libraries include some implementation of BST.

# ABLs in C++: Map and Set

- In C++, the *Map* and *Set* classes are implemented using BSTs
- *Map* Accept Key-value pairs;
- *Set* Accepts only Keys;

# Using Map in C++

```
#include <map>
map<string, int> ages;    ages.clear();

ages["john"] = 40;
ages["billy"] = 39;
ages["andy"] = 29;
ages["steven"] = 42;
ages["felix"] = 33;

// What is the age of andy?
map<string, int>::iterator it = ages.find("andy");
cout << it->second << endl;

// Which names are between "f" and "m" ??
for (map<string, int>::iterator it =
     age.lower_bound("f");                 // finds felix
     it != age.upper_bound("m"); it++)   // finds johm
        cout << " " << ((string)it->first).c_str();
```

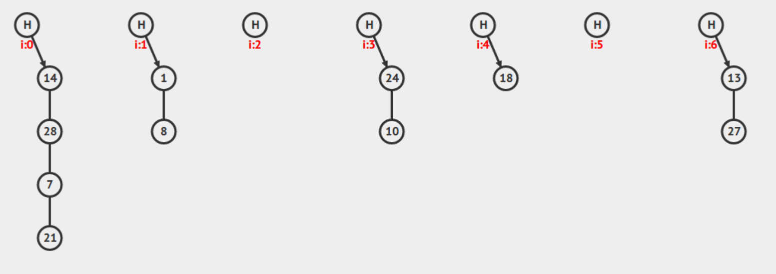
# Using Set in C++

```
#include <set>
set<int> CDs;
CDs.clear();

// Adding some values
CDs.insert(1000); CDs.insert(999); CDs.insert(1337);
CDs.insert(1313); CDs.insert(100020);

// Testing if a particular value exists (O(logn))
set<int>::iterator f = used_values.find(79);
if (f == used_values.end())
  cout << "not found!\n";
else
  cout << *f;    // Index!
```

# Hash Tables




- Very fast insertion and Search – Slow iteration;
- Simple implementation using *unordered_map*;
- Important: Policies about collisions;
- Learn more about hash tables here:
  `https://visualgo.net/ja/hashtable`

# Hand-making Data Structures

- Sometimes, it is necessary to extend the standard data structures (arrays, maps, etc)
- Other times, it is necessary to implement data structures not included in the standard libraries (graphs, UFDS, etc)
- Let's see a few examples.

# Union-Find Disjoint Set (UFDS)

## Motivating Problem

### Network Connections – UVA793

In a network with *n* computers, some are connected to others.

**Input:** A series of “commands”

- c i j – Means computer *i* is connected to computer *j*
- q i j – Question: is computer *i* connected to computer *j*?

**Output:** The number of “q” with answer yes, and the number of “q” with answer no.

# Union-Find Disjoint Set (UFDS)

## Motivating Problem – Naive answer

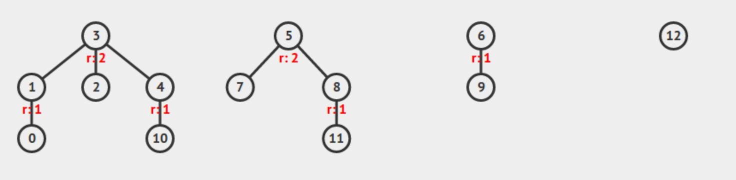
- One idea: Use a Neighborhood Matrix ($n \times n$) initalized with zeros.
- For every “c i j”, $N_{i,j}$, $N_{j,i}$ becomes 1.
- We can follow the graph to answer “q i j”.

How good is this solution?

- Cost to insert a new connection: O(1)
- Cost to check if “q i j”: O(n) (worst case)

We can do better!

# Union-Find Disjoint Set




- The UFDS keeps sets of items, each is represented by a parent;
- When you join two sets You join their parents;
- When you test the parent of an item You flatten the tree;
- Test_item and Join_item are both O(1);
- More Information `https://visualgo.net/ja/ufds`;

# UFDS Implementation using Arrays

```
int p[MAX], r[MAX];
int find(int x) {
    return x == p[x] ? x : p[x]=find(p[x]);
}
int join(int x, int y) {
    x = find(x), y = find(y);
    if(x != y) {
        if(r[x] < r[y])
            p[x] = y, r[y] += r[x];
        else
            p[y] = x, r[x] += r[y];
        return 1;
    }
    return 0;
}
void init() {
    for(int i = 0; i < MAX; i++)
        p[i] = i, r[i] = 1; }
```

# Union Find Disjoint Set

## Problem II – War

From a set of 10k people, some are friends, other are enemies.

- If A,B are friends, and B,C are friends, then A,C are friends
- If A,B are friends, and B,C are enemies, then A,C are enemies
- If A,B are enemies, and B,C are enemies, then A,C are friends

**Input:** A series of commands from the set below:

- SetFriends(i,j)
- SetEnemies(i,j)
- TestFriends(i,j)
- TestEnemies(i,j)

**Output:**

- If a “SetFriends” or “SetEnemies” is impossible, output “-1”
- For a “TestFriends”, “TestEnemies”, output 0 - false, 1 - true

# Union Find Disjoint Set

## Problem II – War

This problem is similar to “Networking”, but now you need to keep track of **TWO** relations.

Some ideas:

- Keep UFDS for friends, and UFDS for enemies?
- Keep an “enemy” flag for each person?
- Add “negative people” to friend-set on UFDS?

Which idea is easier to implement?

# Segment Tree and Fenwick Tree (Self Study)

There are many more specific data structures which are common in programming contests.

Two suggestions for you to study by yourself:

- Segment Tree (section 2.4.3): Finds and update the largest values in intervals in an unordered set.
- Fenwick Tree (section 2.4.4): Finds and update the sum of values in intervals in an unordered set.

# Before ending the class

Final Discussions:

- Problem Hints: File Fragmentation, Grid Successors
- Any Questions?
- Silly Video: Sorting Music

End of the Class!