CPSC 490 – Problem Solving in Computer Science

Lecture 2: Implementation & Debugging, Brute Force, Graph Representations, O(1) Memory Problems

Jason Chiu and Raunak Kumar

2017/01/06

University of British Columbia

Common Bugs and Pitfalls

Last time

- Slow IO
- Overflow / underflow
- Floating point errors

Yet there are many more ways for things to go terribly wrong!

A Few Words About Assignment 1

If you have not started yet, start now!

Everything you need for problems ABCD is in lectures 1 and 2!

A Few Words About Assignment 1

Regarding incorrect submissions

- Before you submit: think about what could possibly go wrong!
  - If everyone gets a problem wrong, either the implementation is bug prone, or there must be a trap!!
- No matter how many times you submit, an algorithm with the wrong time complexity is not going to run in time!
  - Apply CPSC 221/320 skills to estimate Big-O time!
  - Make worse case input and measure execution time!
  - You can assume $\approx 10^8$ operations / second
- Floating point numbers are NASTY!

A Few Words About Assignment 1

Regarding judge environment

- Java: please put main function in public class of the same name as the file name (e.g. `public class A` in `A.java`)
- C++: stack limit is pretty small, so do not use recursion of depth 1000000 and do not declare array of size 1000000 on the stack.
- Judge is run on `annacis.ugrad.cs.ubc.ca`, so if you get compile / run-time error, you may want to ssh to the server and try to run it yourself there.

Problem 1

Input: A single line with $0 \leq n \leq 10^6$ space-separated ints in $[0, 10^6]$.

Output: the $n - 1$ smallest numbers (i.e. all except the largest one), sorted, space-separated, in a single line.

Problem 1

Input: A single line with $0 \leq n \leq 10^6$ space-separated ints in $[0, 10^6]$.

Output: the $n - 1$ smallest numbers (i.e. all except the largest one), sorted, space-separated, in a single line.

What's wrong with this C++ code?

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> v; int x;
    while (cin >> x) v.push_back(x);
    sort(v.begin(), v.end());
    for (int i = 0; i < v.size()-1; i++) {
        cout << v[i] << " ";
    }
    cout << endl;
}
```

Problem 1 – Solution

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    ios_base::sync_with_stdio(0); cin.tie(0);
    vector<int> v; int x;
    while (cin >> x) v.push_back(x);
    sort(v.begin(), v.end());
    for (int i = 0; i+1 < v.size(); i++) {
        if (i > 0) cout << " ";
        cout << v[i];
    }
    cout << endl;
}
```

Problem 1 – Analysis

- Need fast IO
- In C++, sizes are unsigned $\Rightarrow$ type-cast first!
- Be careful of white space

Problem 2

Input: Two lines each with ≤ 100 space-separated ints in $[0, 10^6]$.

Output: For each line in input, output the $n-1$ smallest numbers (i.e. all except the largest one), sorted, space-separated, in a single line.

Problem 2 – “Solution”

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> v; int x; stringstream ss; string line;
    getline(cin, line); ss << line; // 1st line
    while (ss >> x) v.push_back(x);
    sort(v.begin(), v.end());
    for (int i = 0; i+1 < v.size(); i++) {
        if (i>0) cout << " "; cout << v[i];
    }
    cout << endl;

    vector<int> v2; int x2; stringstream ss2; string line2;
    getline(cin, line2); ss2 << line; // 2nd line
    while (ss2 >> x2) v.push_back(x2);
    sort(v2.begin(), v2.end());
    for (int i = 0; i+1 < v2.size(); i++) {
        if (i>0) cout << " "; cout << v[i];
    }
    cout << endl;
}
```

Problem 2 – Solution 1

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> v1; int x1; stringstream ss1; string line1;
    getline(cin, line1); ss1 << line1; // 1st line
    while (ss1 >> x1) v1.push_back(x1);
    sort(v1.begin(), v1.end());
    for (int i = 0; i+1 < v1.size(); i++) {
        if (i>0) cout << " "; cout << v1[i];
    }
    cout << endl;

    vector<int> v2; int x2; stringstream ss2; string line2;
    getline(cin, line2); ss2 << line2; // 2nd line
    while (ss2 >> x2) v2.push_back(x2);
    sort(v2.begin(), v2.end());
    for (int i = 0; i+1 < v2.size(); i++) {
        if (i>0) cout << " "; cout << v2[i];
    }
    cout << endl;
}
```

Problem 2 – Solution 2

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    for (int k = 0; k < 2; k++) {
        vector<int> v; int x; stringstream ss; string line;
        getline(cin, line); ss << line; // 1st line
        while (ss >> x) v.push_back(x);
        sort(v.begin(), v.end());
        for (int i = 0; i+1 < v.size(); i++) {
            if (i>0) cout << " ";
            cout << v[i];
        }
        cout << endl;
    }
}
```

Problem 2 – Analysis

- Try not to duplicate code
- If must duplicate, make sure to change variable names in BOTH copies so compiler can catch errors.
- In C++ `getline()` and `stringstream` are useful for input reading when the size of input is not given.

Problem 3

Input: a partially filled valid sudoku puzzle, with 0 for unfilled cells.

Output: any solution

Problem 3

Input: a partially filled valid sudoku puzzle, with 0 for unfilled cells.

Output: any solution

Why slow?

```
1  bool valid(vector<int> grid, int z, int k) { /*...*/ }
2  void dfs(vector<int> grid, int z) {
3      if (z == 81) { printSol(grid); }
4      for (int k = 1; k <= 9; k++) {
5          if (!valid(grid, z, k)) continue;
6          vector<int> nGrid = grid; nGrid[z] = k;
7          int nZ=z+1; while (nZ<81 && grid[nZ]>0) nZ++;
8          dfs(nGrid, nZ));
9      }
10 }
```

Problem 3 – Solution

```
1  bool valid(vector<int> &grid, int z, int k) { /*...*/ }
2  bool dfs(vector<int> &grid, int z) { // pass by ref.
3      if (z == 81) { printSol(grid); return true; }
4      for (int k = 1; k <= 9; k++) {
5          if (!valid(grid, z, k)) continue;
6
7          grid[z] = k; // modify
8          int nZ=z+1; while (nZ<81 && grid[nZ]>0) nZ++;
9          bool res = dfs(grid, nZ)); // recurse
10         grid[z] = 0; // reset
11
12         if (res) return true; // break early
13     }
14     return false;
15 }
```

Problem 3 – Analysis

- Pass states by reference
- Modify states in place
- Break early in DFS

Problem 4

Input: $4 \leq L \leq 100$ sets of integers from $[1, 64]$

Output: the size of the biggest set that can be made by taking union of 4 disjoint sets from the given list.

Problem 4 – Solution (too slow)

```
#include <bits/stdc++.h>
using namespace std;
int main() { int L; vector<vector<int>> sets;
  /* assume input read into L and sets */
  int best = 0; bool seen[64];
  for (int i1 = 0;    i1 < L-3; i1++) {
  for (int i2 = i1+1; i2 < L-2; i2++) {
  for (int i3 = i2+1; i3 < L-1; i3++) {
  for (int i4 = i3+1; i4 < L;   i4++) {
    bool good = 1; memset(seen, 0, sizeof seen);
    for (int v : sets[i1]) seen[v]=true;
    for (int v : sets[i2]){ if(seen[v]){good=0;break;} seen[v]=1; }
    for (int v : sets[i3]){ if(seen[v]){good=0;break;} seen[v]=1; }
    for (int v : sets[i4]){ if(seen[v]){good=0;break;} seen[v]=1; }
    if (good)
        best = max(best, sets[i1].size()+sets[i2].size()+
                         sets[i3].size()+sets[i4].size());
  }}}}
  cout << best << endl;
  return 0;
}
```

Problem 4 – Solution (fast!)

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() { int L; unsigned long long sets[100];
  /* assume input read into L and into sets as bitmask */
  int best = 0;
  for (int i1 = 0; i1 < L-3; i1++) {
    auto a = sets[i1];
    for (int i2 = i1+1; i2 < L-2; i2++) {
      auto b = sets[i2]; if (b&a) continue;
      for (int i3 = i2+1; i3 < L-1; i3++) {
        auto c = sets[i3]; if ((c&a) || (c&b)) continue;
        for (int i4 = i3+1; i4 < L; i4++) {
          auto d = sets[i4]; if ((d&a) || (d&b) || (d&c)) continue;
          best = max(best, __builtin_popcount(a|b|c|d));
  }}}}
  cout << best << endl;
  return 0;
}
```

Problem 4 – Analysis

- Use and abuse 64-bit and 128-bit integers for 50-100x speedup!
  - Set operations: union = `|`, intersection = `&`
  - Bit-shift = multiply / divide by power of 2
  - Remember to use unsigned if using most significant bit
  - `x & -x` gives the least significant 1 in x (why?)
- g++ has many built-in bit-twiddling functions
  - `__builtin_popcount` / `__builtin_popcountll`: count 1 bits
  - `__builtin_ctz` / `__builtin_ctzll`: count trailing zeros
  - `__builtin_clz` / `__builtin_clzll`: count leading zeros
- Caution: <, >, ==, etc have **higher** precedence than bitwise operators (&, |, ^), so always use parentheses.

We will come back to bitmasks later in the course!

Other Common Bugs

Input / Output

- Read input incorrectly (spaces in string, 32 vs. 64-bit)
- Forget to read all input
- Forget to print newline

General

- Forget to reset variables per test case
- Array index out of bounds
  - Always make array size slightly bigger to avoid this
- Aliasing: declare local variable with same name as global
- Custom comparator ill-defined
- Forget to save file / submit wrong file / select wrong language

Debugging Tips

- Turn on all compile warnings (lots of code that look like they shouldn't compile actually do!)
- Use print statements, debugger to trace execution
- Valgrind is very helpful for memory errors
- Make hard test cases (sample input is often deceptive)
- Run your code in judge environment

Optimizations

- Use an asymptotically faster algorithm
- Optimize constant factors
  - Memory allocation / reset
  - Copying data structures
  - Complicated data structures: e.g. `std::set`
  - Cache locality: array vs. hash map
  - Branch misprediction
  - Java: Object allocation, immutable vs. mutable
  - `int`/`float` vs. `long long`/`double`
  - Bitmasks

Graphs

What are graphs?

- Represent relationships with nodes (= vertices) and edges.
- Notation: Call the set of nodes V, and the set of edges E.
- Each edge connects some pair of nodes, edges can be
  - Bidirectional (undirected)
  - One-directional (directed)
  - Weighted (all weights are 1 for now)
- A tree is a connected graph with $|V| - 1$ edges

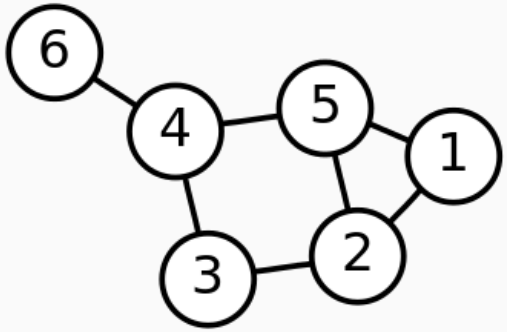

Figure 1: An undirected graph (Source: Wikipedia)

Why study graphs?

Tons of problems we can solve in terms of graphs!

- Shortest path problems
- Dependency/connectivity problems
- Cycle detection
- Minimum cost substructures (trees, matchings, etc.)
- and many many more...

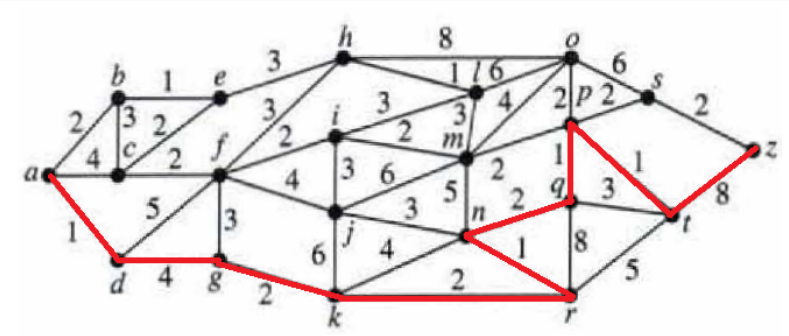

Figure 2: Shortest path problems have a natural representation in graphs.

Graph Representations

Adjacency List

- Idea: for each node, store array of neighbor nodes
- C++
  - `vector<int> adj[N];`
  - `adj[u].push_back(v);`
- Java
  - `ArrayList<Integer>[] adj = new ArrayList[N];`
  - `for(int i=0; i<N; i++) adj[i]=new ArrayList<>();`
  - `adj[u].add(v);`
- Advantages: memory efficient, easy to iterate through neighbors
- Disadvantages: hard to find if (u, v) is an edge
  - Can be fixed by using a hash map instead, but slower

Graph Representations

Adjacency Matrix

- Idea: *adjmat*[*u*][*v*] is true if there's an edge ($u \to v$)
- C++
  - `bool adjmat[N][N]; memset(adj, 0, sizeof adjmat);`
  - `adjmat[u][v] = true;`
- Java
  - `int[][] adjmat = new int[N][N];`
  - `adjmat[u][v] = true;`
- Advantages: fast for dense graphs, easy to test if (u, v) is edge
- Disadvantages: hard to iterate through neighbors, not efficient for sparse graphs
  - Can be fixed by using a hash map instead, but slower

Watch out!

Common edge cases and bugs with graphs and graph algorithms

- Disconnected graph
- Graph with zero / one vertices / edges
- Directed vs. undirected graph
- Multiple edges between same pair of vertices
- Self-loop
- Edges of zero / negative weight
- Indexing errors: `grid[x][y]` or `grid[y][x]`?

Watch out!

Common edge cases and bugs with trees

- Multiple trees (i.e. a forest)
- Tree with zero or one node
- Tree that is a very long line
- Tree with very big breadth
- “Broomstick” tree: above two cases combined

O(1) memory problems

Problem 5

Detect if there is a loop in linked list in $O(n)$ time and $O(1)$ memory.

```
1  struct node {
2      node * nxt;
3      int value;
4  }
```

Problem 5

Detect if there is a loop in linked list in $O(n)$ time and $O(1)$ memory.

```
1  struct node {
2      node * nxt;
3      int value;
4  }
```

Possible solutions:

1. Destroy the pointers as you go
2. Use the last bit of the nxt pointer as visited flag
3. Guess max(arm size, loop size); double the guess if it fails
4. Walk two pointers, one twice as fast as another, see if they meet

Problem 6

Traverse a binary tree in $O(n)$ time and $O(1)$ memory.

```
1  struct node {
2      node * left, right;
3      int value;
4  }
```

Problem 6

Traverse a binary tree in $O(n)$ time and $O(1)$ memory.

```
1  struct node {
2      node * left, right;
3      int value;
4  }
```

Solution hint: use previous node as a state to figure out whether we should go to left child, right child, or to parent.

Next Class

- Breadth-first search
- Shortest-path: Dijkstra
- Minimum spanning tree