Compilando Conocimiento

Refence

Competitive Programming

Rosas Hernandez Oscar Andrés

July 2018

Contents

Part I

Things to Learn / To Do

Chapter 1

C++

1.1 Integrals

1.1.1 int vs long vs long long

```
int minValue {-2,147,483,648};
int maxValue {2,147,483,647};

long long minValue {-9,223,372,036,854,775,808};
long long maxValue {9,223,372,036,854,775,807};

unsigned int maxValueIntUnsigned {4,294,967,295};
unsigned long long maxValueLLUnsigned
    {18,446,744,073,709,551,615};
```

1.1.2 Fixed width (int32_t, uint64_t, ...)

```
#include <cstdint>

int8_t likeChar {};
int16_t likeShort {};
int32_t likeInt {};
int64_t likeLong {};

// And the unsigned versions:
uint8_t likeChar {};
uint16_t likeShort {};
uint32_t likeInt {};
uint64_t likeLong {};
```

1.1.3 Bits

1.1.4 Fast I / O

```
// No merge cin & cout with scanf & printf
ios::sync_with_stdio(false);

// No merge cin / cout
cin.tie(nullptr);
```

```
template <class T>
inline void getNumberFast(T &result) {
    T number {};
    T sign {1};

    char currentDigit {getchar_unlocked()};

    while(currentDigit < '0' or currentDigit > '9') {
        currentDigit = getchar_unlocked();
        if (currentDigit == '-')  sign = -1;
    }

    while ('0' <= currentDigit and currentDigit <= '9') {
        number = (number << 3) + (number << 1);
        number += currentDigit - '0';
        currentDigit = getchar_unlocked();
    }

    if (sign) result = -number;
    else result = number;
}
```

Part II

Number Theory

Chapter 2

Primes

2.1 Sieve of Eratosthenes

2.1.1 Get the Boolean Version

```
template<typename T>
auto getIsPrime(T maxValue) -> std::vector<bool> {
    std::vector<bool> isPrime (maxValue + 1, true);
    isPrime[0] = isPrime[1] = false;

    for (T i {4}; i <= maxValue; i += 2) isPrime[i] = false;

    for (T i {3}; i * i <= maxValue; i += 2) {
        if (not isPrime[i]) continue;

        T multiple {i * i}, step {2 * i};
        while (multiple <= maxValue) {
            isPrime[multiple] = false;
            multiple += step;
        }
    }

    return isPrime;
}
```

2.1.2 Get the Vector of Primes

```
template<typename T>
auto getPrimes(T maxValue) -> std::vector<T> {
    std::vector<bool> isPrime (maxValue + 1, true);
    std::vector<T> primes {2};

    // Just to do it if you need the bools too.
    // isPrime[0] = isPrime[1] = false;
    // for (T i = 4; i <= n; i += 2) isPrime[i] = false;

    for (T i {3}; i <= maxValue; i += 2) {
        if (not isPrime[i]) continue;
        primes.push_back(i);

        T multiple {i * i}, step {2 * i};
        while (multiple <= maxValue) {
            isPrime[multiple] = false;
            multiple += step;
        }
    }

    return primes;
}
```

Part III

Graphs

Chapter 3

Simple Graphs

3.1 GraphRepresentations

3.1.1 GraphAdjacencyList

```
#include <vector>

using namespace std;

template <typename nodeID, typename fn>
class GraphAdjacencyList {
 private:
  std::vector<std::vector<nodeID>> adjacencyLists;

 public:
  const bool isBidirectional;

  GraphAdjacencyList(nodeID numOfNodes, bool isBidirectional
    = true)
      : isBidirectional(isBidirectional),
    adjacencyLists(numOfNodes) {}

  void addEdge(nodeID fromThisNode, nodeID toThisNode) {
    adjacencyLists[fromThisNode].push_back(toThisNode);
    if (not isBidirectional) return;
    adjacencyLists[toThisNode].push_back(fromThisNode);
  }

  void addConections(const vector<pair<nodeID, nodeID>>&
    conections) {
    for (const auto& edge : conections) addEdge(edge.first,
    edge.second);
  }

  void show() {
    nodeID node {};
    for (auto& adjacencyList : adjacencyLists) {
      cout << "Node ID = " << node++ << ": [";
      for (auto& node : adjacencyList) cout << node << " ";
      cout << "]" << '\n';
    }
  }

  auto BFS(nodeID initialNode, fn functionToCall) -> void;
  auto DFS(nodeID initialNode, fn functionToCall) -> void;
};
```

3.1.2 PonderateGraph

```
#include <set>

template <typename nodeID, typename weight>
struct node {
  nodeID from, to;
  weight cost;
};

template <typename nodeID, typename weight>
class PonderateGraph {
 private:
  std::vector<node<nodeID, weight>> edges;

 public:
  void addEdge(nodeID fromThisNode, nodeID toThisNode,
    weight cost) {
    edges.push_back({fromThisNode, toThisNode, cost});
```

```
  }

  auto KruskalMinimumExpansionTree(nodeID maxNodeID)
      -> std::pair<set<nodeID>, weight>;
};
```

3.2 UnionFind - Dijoined set

3.2.1 Simple UnionFind

```
#include <iostream>
#include <numeric>
#include <vector>

class SimpleUnionFind {
 private:
  std::vector<int> nodesInComponent, parent;

 public:
  SimpleUnionFind(int n) : nodesInComponent(n, 1) {
    parent.resize(n);
    while (--n) parent[n] = n;
  }

  auto findParentNode(int u) -> int {
    if (parent[u] == u) return u;
    return parent[u] = findParentNode(parent[u]);
  }

  auto existPath(int u, int v) -> bool {
    return findParentNode(v) == findParentNode(u);
  }

  auto numberOfElementsInAComponent(int u) -> int {
    return nodesInComponent[findParentNode(u)];
  }

  auto joinSets(int u, int v) -> void {
    int setU = findParentNode(u), setV = findParentNode(v);
    if (setU == setV) return;

    parent[setU] = setV;
    nodesInComponent[setV] += nodesInComponent[setU];
  }
};
```

3.2.2 Real UnionFind

```
#include <map>
#include <unordered_map>

/**
 *
 * You have many nodes (with ID's as numbers) and the nodes
   are connected (ie,
 * node 2 with node 4, 5, 8) Use UnionFind to find if 2
   nodes are connected
 * or how many nodes are in a connected to a given node.
 *
 */
template <typename parentContainer, typename ID = int,
    typename numCount = int,
          typename numRank = int>
class UnionFind {
 private:
  parentContainer parent;
  std::vector<numCount> nodesInComponent;
  std::vector<numRank> rank;

  // Get the representant node ID from a component
  auto findParentNode(ID node) -> ID {
    ID& nodeParent = parent[node];
    if (node == nodeParent) return node;

    nodeParent = findParentNode(nodeParent);
    return nodeParent;
  }

 public:
  UnionFind(ID numNodes) : nodesInComponent(numNodes, 1),
    rank(numNodes, 0) {
    parent.resize(numNodes);  // Delete if parentContainer
    is a map
    while (--numNodes) parent[numNodes] = numNodes;
  }

  auto existPath(ID nodeA, ID nodeB) -> bool {
    return findParentNode(nodeA) == findParentNode(nodeB);
  }

  auto numberOfElementsInAComponent(ID node) -> numCount {
    return nodesInComponent[findParentNode(node)];
  }
```

```
  auto joinComponent(ID nodeA, ID nodeB) -> void {
    ID setA {findParentNode(nodeA)}, setB
    {findParentNode(nodeB)};

    if (setA == setB) return;
    if (rank[setA] < rank[setB]) std::swap(setA, setB);

    parent[setB] = setA;
    nodesInComponent[setA] += nodesInComponent[setB];

    if (rank[setA] == rank[setB]) ++rank[setA];
  }
};
```