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

# Data Structures

## 3.1 Fenwick Tree

```
#include <functional>
#include <vector>

#include <iostream>

using std::cin;
using std::cout;
using std::endl;

/**
 *
 * You have an array (starting with 0 or you can use
   buildFromArray),
 * you can use FenwickTree to get the sum of all elements in
   a range
 * also, you can increase a position by a value
 *
 */
template <typename element = int, typename index = int>
class FenwickTree {
 private:
  const int MAX_SIZE;
  std::vector<element> bit {};

  static auto getNext(index i) -> index { return i | (i +
    1); }

 public:
  FenwickTree(int MAX_SIZE = 100000) : MAX_SIZE {MAX_SIZE},
    bit(MAX_SIZE, 0) {}
auto buildFromArray(const std::vector<element>& data) ->
  void {
   for (index i {}; i < MAX_SIZE; ++i) {
     bit[i] = bit[i] + data[i];
     const auto nextIndex {getNext(i)};
     if (nextIndex < MAX_SIZE) bit[nextIndex] = bit[i] +
  bit[nextIndex];
   }
}

// get the sum from [0, end]
auto sum(int end) -> element const {
   element answer {};
   while (end >= 0) {
     answer = answer + bit[end];
     end = (end & (end + 1)) - 1;
   }
   return answer;
}

// get the sum from [start, end]
auto sum(index start, index end) -> element const {
   return sum(end) - sum(start - 1);
}

// increase the position by a value
auto increase(index position, element value) -> void {
   while (position < MAX_SIZE) {
     bit[position] = bit[position] + value;
     position = getNext(position);
   }
}
```

```
  void showArray() {
    cout << "[";
    for (int i {}; i < MAX_SIZE; ++i) cout << sum(i, i) <<
   ", ";
    cout << "]" << endl;
  }

  void showPrefixArray() {
    cout << "[";
    for (int i {}; i < MAX_SIZE; ++i) cout << sum(i) << ", ";
    cout << "]" << endl;
  }
};

int main() {
  const int sizeOfRange {5};
  auto f = FenwickTree<> {sizeOfRange};
  f.increase(0, 4);
  f.showArray();
  f.showPrefixArray();

  cout << f.sum(0, 4) << endl;

  return 0;
}
```

# Chapter 4

# Simple Graphs

## 4.1 GraphRepresentations

### 4.1.1 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;
};
```

### 4.1.2 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:
  auto addEdge(nodeID fromThisNode, nodeID toThisNode,
    weight cost) -> void {
    edges.emplace_back({fromThisNode, toThisNode, cost});
```

```
  }

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

## 4.2 BFS

```
#include <iostream>
#include <queue>
#include <stack>
#include <vector>

#include "GraphRepresentations.cpp"

template <typename nodeID, typename fn>
auto GraphAdjacencyList<nodeID, fn>::BFS(nodeID initialNode,
   fn functionToCall) -> void {
  std::vector<bool> visited(adjacencyLists.size(), false);
  std::queue<int> nodesToProcess({initialNode});

  while (not nodesToProcess.empty()) {
    auto node {nodesToProcess.front()};
    nodesToProcess.pop();

    if (not visited[node]) {
      functionToCall(node, visited);
      visited[node] = true;
    }

    for (auto& adjacentNode : adjacencyLists[node])
      if (not visited[adjacentNode])
   nodesToProcess.push(adjacentNode);
  }
}
```

## 4.3 DFS

```
#include <iostream>
#include <queue>
#include <stack>
#include <vector>

#include "GraphRepresentations.cpp"

template <typename nodeID, typename fn>
auto GraphAdjacencyList<nodeID, fn>::DFS(nodeID initialNode,
   fn functionToCall) -> void {
  std::vector<bool> visited(adjacencyLists.size(), false);
  std::stack<int> nodesToProcess({initialNode});

  while (not nodesToProcess.empty()) {
    auto node {nodesToProcess.top()};
    nodesToProcess.pop();

    if (not visited[node]) {
      functionToCall(node, visited);
      visited[node] = true;
    }

    for (auto& adjacentNode : adjacencyLists[node])
      if (not visited[adjacentNode])
   nodesToProcess.push(adjacentNode);
  }
}
```

## 4.4 UnionFind - Disjoined set

### 4.4.1 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 joinComponents(int u, int v) -> void {
    int setU = findParentNode(u), setV = findParentNode(v);
    if (setU == setV) return;

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

### 4.4.2 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 joinComponents(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];
  }
};
```

## 4.5 UnionFind - Disjoined set

### 4.5.1 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;
```

```cpp
  }

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

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

## 4.6 Kruskal: Minimum Spanning Tree

```cpp
#include <algorithm>
#include <set>
#include "GraphRepresentations.cpp"
#include "UnionFind.cpp"

template <typename nodeID, typename weight>
auto PonderateGraph<nodeID,
    weight>::KruskalMinimumExpansionTree(
    nodeID nodesInGraph) -> std::pair<set<nodeID>, weight> {
  using node = const node<nodeID, weight>;

  auto minimumSpanningTreeWeight = weight {};
  auto nodesInTree = set<nodeID> {};
  auto graphInfo = UnionFind<std::vector<nodeID>, nodeID>
    {nodesInGraph};
  auto sortNode = [](node& n1, node& n2) { return n1.cost <
    n2.cost; };
  sort(edges.begin(), edges.end(), sortNode);

  for (node& edge : edges) {
    // check if edge is creating cycle
    if (graphInfo.existPath(edge.to, edge.from)) continue;

    nodesInTree.insert(edge.to);
    nodesInTree.insert(edge.from);

    minimumSpanningTreeWeight += edge.cost;
    graphInfo.joinComponents(edge.to, edge.from);
    if (graphInfo.numberOfElementsInAComponent(edge.to) ==
   nodesInGraph) break;
  }

  return {nodesInTree, minimumSpanningTreeWeight};
}
```