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 Max & min

```
#include <limits>       // std::numeric_limits

int main () {
    const auto minVal = std::numeric_limits<int>::min();
    const auto maxVal = std::numeric_limits<int>::max();
    return 0;
}
```

### 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);
cout.tie(nullptr);
```

#### Fast input of numbers

```
#include <cstdio>

template <class number>
inline auto getNumberFast() -> number {
  auto result = number {};
  auto isNegative = false;
  auto currentDigit = char {getchar_unlocked()};

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

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

  return isNegative ? -result : result;
}
```

## 1.2 Input

### 1.2.1 Precision

```
#include <iomanip>
#include <iostream>

auto main() -> int {
    using namespace std:

    auto f = 301.14159;
    cout << std::fixed;

    cout << setprecision(5) << f << '\n';   // 301.14159
    cout << setprecision(8) << f << '\n';   //  301.14159000

    return 0;
}
```

### 1.2.2 Base

```
#include <iostream>     // std::cout, std::endl
#include <iomanip>      // std::setbase

int main () {
    using namespace std:
    cout << std::setbase(16) << 110 << endl;    //6e
    return 0;
}
```

### 1.2.3 Read list of unknow data

```
#include <iostream>
#include <sstream>
#include <vector>

auto main() -> int {
    auto buffer = std::string {};
    while (getline(std::cin, buffer)) {
        auto bufferStream = std::istringstream {std::move(buffer)};

        auto list = std::vector<int> {};
        auto num = int {};
        while (bufferStream >> num) list.push_back(num);

        ... (use list)
    }

return 0;
}
```

# Part II

# Number Theory

# Chapter 2

# General

## 2.1 Binary Exponentiation

```
  while (exponent > 0) {
    if (exponent & 1) solution *= base;

    base = base * base;
    exponent = exponent >> 1;
  }

  return solution;
}

template <typename integer>
auto modularBinaryExponentation(integer base, integer exponent, integer mod)
    -> integer {
```

## 2.2 Modular Binary Exponentiation

```
  base = base % mod;

  while (exponent > 0) {
    if (exponent & 1) solution = (base * solution) % mod;

    base = (base * base) % mod;
    exponent = exponent >> 1;
  }

  return solution;
}
```

# Chapter 3

# Primes

## 3.1 Sieve of Eratosthenes

### 3.1.1 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;
}
```

### 3.1.2 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 4

# Data Structures

## 4.1 Segment Tree

```cpp
template <typename T = int>
struct segment_tree {
  const int n;
  const T neutral;
  T (*fn)(T, T);
  vector<T> nodes;
  segment_tree(int n, T fn(T, T), T neutral)
      : nodes(2 * n, neutral), n(n), fn(fn), neutral(neutral) {}

  auto build(const vector<T>& data) -> void {
    for (auto i = 0; i < data.size(); ++i) nodes[n + i] = data[i];
    for (auto i = n - 1; i > 0; --i) nodes[i] = fn(nodes[2 * i], nodes[2 * i
    + 1]);
  }

  auto update(const int p, const T value) -> void {
    nodes[n + p] = value;
    for (auto i = p + n; i > 1; i /= 2) nodes[i / 2] = fn(nodes[i xor 0],
     nodes[i xor 1]);
  }

  auto query(const int left, const int right) -> T const {
    auto result = neutral;
    for (auto l = left + n, r = right + n + 1; l < r; l /= 2, r /= 2) {
      if (l bitand 1) result = fn(result, nodes[l++]);
      if (r bitand 1) result = fn(result, nodes[--r]);
    }
    return result;
  }
};
```

```cpp
template <typename T = int>
struct segment_tree {
 private:
  using node = int;
  const int n;
  const T neutral;
  T (*fn)(T, T);
  vector<T> nodes;

  void build(const vector<T>& data, int begin, int end, node current) {
    if (begin == end) {
      nodes[current] = data[begin];
      return;
    }

    auto middle = (begin + end) / 2;
    build(data, begin, middle, current * 2 + 1);
    build(data, middle + 1, end, current * 2 + 2);

    nodes[current] = fn(nodes[current * 2 + 1], nodes[current * 2 + 2]);
  }

  T query(int begin, int end, node current, int left, int right) {
    if (end < left or right < begin) return neutral;
    if (begin >= left and end <= right) return nodes[current];

    auto middle = (begin + end) / 2;
    auto l = query(begin, middle, current * 2 + 1, left, right);
    auto r = query(middle + 1, end, current * 2 + 2, left, right);

    return fn(l, r);
  }

  void update(int begin, int end, node current, int index, const T& val) {
    if (end < index or index < begin) return;

    if (begin == index and end == index) {
      nodes[current] = val;
      do {
        current = (current - 1) / 2;
        nodes[current] = fn(nodes[current * 2 + 1], nodes[current * 2 + 2]);
      } while (current != 0);

      return;
    }

    auto middle = (begin + end) / 2;
    update(begin, middle, current * 2 + 1, index, val);
    update(middle + 1, end, current * 2 + 2, index, val);
  }
```

```
 public:
  segment_tree(int n, T fn(T, T), T neutral)
      : nodes(4 * n, neutral), n(n), fn(fn), neutral(neutral) {}

  void build(const vector<T>& data) { build(data, 0, n - 1, 0); }
  T query(int left, int right) { return query(0, n - 1, 0, left, right); }
  void update(int index, const T& val) { update(0, n - 1, 0, index, val); }
};
```

## 4.2 Fenwick Tree

```
template <typename T = int>
class fenwick_tree {
 private:
  const int n;
  vector<T> nodes;

  static auto get_next(const int i) -> int { return i bitor (i + 1); }

 public:
  fenwick_tree(int n) : n(n), nodes(n, 0) {}

  auto build(const vector<T>& data) -> void {
    for (auto i = 0; i < n; ++i) {
      nodes[i] = nodes[i] + data[i];
      const auto next_index = get_next(i);
      if (next_index < n) nodes[next_index] = nodes[i] + nodes[next_index];
    }
  }

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

  auto sum(const int start, const int end) -> T const {
    return sum(end) - sum(start - 1);
  }

  auto increase(const int position, const T value) -> void {  // increase the
     position by a val
    for (auto p = position; p < n; p = get_next(p))
      nodes[p] = nodes[p] + value;
  }
};
```

## 4.3 UnionFind - Disjoined set

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

/**
 * You have many nodes connected (ie, node 2 with 4 and 8).
 * Use UnionFind to find if 2 nodes are connected or and
 * how many nodes can I go to from a given node.
 */
template <typename id = int>
class union_find {
  vector<id> num_connected_nodes_of, parent_of, rank_of;

 public:
  union_find(const id num_nodes)
      : num_connected_nodes_of(num_nodes, 1), parent_of(num_nodes),
     rank_of(num_nodes, 0) {
    for (id i = 0; i < num_nodes; ++i) { parent_of[i] = i; }
  }

  auto get_component_id(const int u) -> int {
    if (parent_of[u] != u) { parent_of[u] = get_component_id(parent_of[u]); }
    return parent_of[u];
  }

  auto join_components(const id u, const id v) -> void {
    auto id_of_u = get_component_id(u), id_of_v = get_component_id(v);
    if (id_of_u == id_of_v) return;

    if (rank_of[id_of_u] > rank_of[id_of_v]) swap(id_of_u, id_of_v);

    parent_of[id_of_u] = parent_of[id_of_v];
    num_connected_nodes_of[id_of_v] += num_connected_nodes_of[id_of_u];
    if (rank_of[id_of_u] == rank_of[id_of_v]) ++rank_of[id_of_v];
  }

  auto num_elements_connected_to(const id u) -> int {
    return num_connected_nodes_of[get_component_id(u)];
  }
};
```

# Chapter 5

# Simple Graphs

## 5.1 GraphRepresentations

### 5.1.1 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>;
};
```

### 5.1.2 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;
};
```

## 5.2 BFS

```
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);
  }
}
```

## 5.3 DFS

```
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);
  }
}
```

## 5.4 Kruskal: Minimum Spanning Tree

```
#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};
}
```

# Part IV

# Bits

# Chapter 6

# Bit manipulation

## 6.1 Shifts

- $x << y = x * 2^y$
- $x >> y = \left\lfloor \frac{x}{2^y} \right\rfloor$

## 6.2 Count on bits

```
// int
__builtin_popcount(n);
// long long
__builtin_popcountll(n);
```

## 6.3 Know if k-th bit is on

```
bool is_on(int number, int place) {
    return (number bitand (1 << place));
}
```

## 6.4 Turn on k-th bit

```
int toogle_on_bit(int number, int place) {
  return (number bitor (1 << place));
}
```

## 6.5 Turn off k-th bit

```
int toogle_off_bit(int number, int place) {
  return (number bitand ~(1 << place));
}
```

## 6.6 Toggle k-th bit (from off to on and viceversa)

```
int toggle_on_bit(int number, int place) {
  return (number xor (1 << k));
}
```

## 6.7 Get the LSB

```
bool lowest_on_bit(int number) {
    return (number bitand (-number));
}
```

## 6.8 Turn on the first K bits

```
int set_first_K_bits(int place) {
  return ((1 << place) - 1);
}
```

# Part V

# Geometry

# Chapter 7

# Data Structures

## 7.1 Options

```
enum class result {
  no_points = 0,
  one_points = 1,
  infinity_points = -1,
};
```

## 7.2 Decimnal

```
#include <cmath>
#include <iostream>

template <typename number = long double>
struct decimal {
  static constexpr number epsilon = 1e-9;
  number x;
  decimal(number x = 0) : x(x) {}

  decimal operator+(const decimal &p) const { return {x + p.x}; }
  decimal operator-(const decimal &p) const { return {x - p.x}; }
  decimal operator*(const decimal &p) const { return {x * p.x}; }
  decimal operator/(const decimal &p) const { return {x / p.x}; }

  decimal operator+=(const decimal &p) { *this = *this + p; return *this; }
  decimal operator-=(const decimal &p) { *this = *this - p; return *this; }
  decimal operator*=(const decimal &p) { *this = *this * p; return *this; }
  decimal operator/=(const decimal &p) { *this = *this / p; return *this; }

  bool operator==(const decimal &p) const { return abs(x - p.x) <= epsilon; }
  bool operator!=(const decimal &p) const { return not(*this == p); }
  bool operator<(const decimal &p) const { return p.x - x > epsilon; }
  bool operator>(const decimal &p) const { return x - p.x > epsilon; }

  bool operator>=(const decimal &p) const { return x - p.x >= -epsilon; }
  bool operator<=(const decimal &p) const { return p.x - x >= -epsilon; }

  int sign() {
    if (x > 0) return 1;
    if (x < 0) return -1;
    return 0;
  }

  friend std::istream &operator>>(std::istream &is, const decimal &p) {
    return is >> p.x; }
  friend std::ostream &operator<<(std::ostream &os, const decimal &p) {
    return os << p.x; }
};
```

## 7.3 Points

```
template <typename number>
struct point {
  number x, y;

  point(number x = 0, number y = 0) : x {x}, y {y} {}

  point operator+(const point& p) const { return {x + p.x, y + p.y}; }
  point operator-(const point& p) const { return {x - p.x, y - p.y}; }
  point operator*(number k) const { return {x * k, y * k}; }
  point operator/(number k) const { return {x / k, y / k}; }

  point operator+=(const point& p)  { *this = *this + p; return *this; }
  point operator-=(const point& p)  { *this = *this - p; return *this; }
  point operator*=(const number& p) { *this = *this * p; return *this; }
  point operator/=(const number& p) { *this = *this / p; return *this; }

  bool operator==(const point& p) const { return x == p.x and y == p.y; }
  bool operator!=(const point& p) const { return not(*this == p); }

  bool operator<(const point& p) const { return x != p.x ? x < p.x : y < p.y;
    }
  bool operator>(const point& p) const { return x != p.x ? x > p.x : y > p.y;
    }

  bool operator<=(const point& p) const { return (*this == p) or *this < p; }
  bool operator>=(const point& p) const { return (*this == p) or *this > p; }

  friend std::istream& operator>>(std::istream& is, point& p) {
    return is >> p.x >> p.y;
  }
```

```cpp
  friend std::ostream& operator<<(std::ostream& os, const point& p) {
    return os << "(" << p.x << ", " << p.y << ")";
  }

  // Operations
  auto norm() -> number const { return { x * x + y * y }; }
  auto length() -> number const { return sqrtl(norm()); }

  auto perpendicular() -> point const { return {-y, x}; }
  auto unit() -> point const { return (*this) / length(); }

  auto rotate(number angle) -> point const {
    return {x * cos(angle) - y * sin(angle), x * sin(angle) + y * cos(angle)};
  }

};
```

```cpp
template <typename T>
auto dot(const point<T>& p, const point<T>& q) -> point<T> {
  return {p.x * p.x + p.y * p.y};
}

template <typename T>
auto cross(const point<T>& p, const point<T>& q) -> T {
  return {p.x * q.y - p.y * q.x};
}

template <typename T>
auto area(const point<T>& a, const point<T>& b, const point<T>& c) -> T {
  const auto ab = b - a, ac = c - a;
  return 0.5 * cross(ab, ac);
}
```