ACM Cheatsheet

Powers of 2

Powers of 2	Value	Powers of 10
2^0	1	10^0
2^1	2	
2^2	4	
2^3	8	10^1
2^4	16	10^0
2^5	32	10^0
2^6	64	10^0
2^7	128	10^3
2^8	256	10^0
2^9	512	10^0
2^{10}	1024	1000^1
2^{20}	1048576	1000^2
2^{30}	1073741824	1000^3
2^{40}		1000^4
2^{50}		1000^5
2^{60}		1000^6

Bit Operations

Checking if ith bit set:

```
int N = 3; // 0 1 1
for (int i = 0; i < 3; ++i) {
    int T = N & (1 << i);
    cout << i << "␣bit␣is␣" << T << "\n";
}
```

Outputs:

```
0 bit is 1
1 bit is 2
2 bit is 0
```

Switch on ith Bit

```
// Bitwise OR: to turn on the jth element.
// 100010
// 001000 (0-indexed)
S = 34;
S |= (1 << 3);    // i = 3
cout << S << "\n";
```

```
42
```

Switch off ith Bit

```
// Turn off the jth element
// 101010
// 111101
S = 42;
T = S & ~(1 << 1);    // i = 1
cout << T << "\n";
```

```
40
```

Flipping ith Bit

```
// flip bit at jth position:
S = 42;
S ^= (1 << 2);    // i = 2
cout << S << "\n";
```

```
46
```

Bit Multiplication

```
// Multiplication by 2: shifting the bits to the left
int S = 7;
S = S << 1;
cout << S << "\n";
```

```
14
```

Bit Division

```
// Dividing by 2: Shifting bits to the right
S = 7;
S = S >> 1;
cout << S << "\n";
```

```
3
```

Position of Least Significant Bit

```
S = 3;    // 0 1 1
T = (S & (-S));
cout << T << "\n";
```

```
1    // 2^0
```

Vector

Ascending sort

```
sort(points.begin(), points.end());
```

Special sort (e.g. descending)

```
bool pairCompare(const pair<K::FT, K::Point_2>& lhs, const pair<K::FT, K::Point_2>& rhs) {
    return lhs.first > rhs.first;
}
sort(points.begin(), points.end(), pairCompare);
```

Min Heap

```
priority_queue<int, vector<int>, greater<int> > min_heap;    // integers
priority_queue<pair<long long, int>, vector<pair<long long, int> >, greater<pair<long long, int> > ↙
    ↳ > min_heap;    // pairs
```

BGL Cheatsheet

General Includes

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/max_cardinality_matching.hpp>
#include <boost/graph/boyer_myrvold_planar_test.hpp>
```

Kruskal MST

```
#include <boost/graph/kruskal_min_spanning_tree.hpp>
vector<Edge> spanning_tree;
kruskal_minimum_spanning_tree(g, std::back_inserter(spanning_tree));
```

Dijkstra Shortest Path

```
#include <boost/graph/dijkstra_shortest_paths.hpp>
// edge weight need to be set!
vector<int> d(N);
vector<Vertex> p(N);
dijkstra_shortest_paths(g, s, predecessor_map(&p[0]).distance_map(&d[0]));
```

Undirected Graph

```
typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property > Graph;
```

Directed Graph

```
typedef adjacency_list<vecS, vecS, directedS, no_property, no_property> Graph;
```

Strong Components

```
#include <boost/graph/strong_components.hpp>
vector<int> scc(N);
int nscc = strong_components(g, &scc[0]);
```

Planarity testing

Eulers formula: $F + V - E = 2$

```
#include <boost/graph/boyer_myrvold_planar_test.hpp>
typedef adjacency_list<vecS, vecS, undirectedS, no_property, no_property> Graph;
boyer_myrvold_planarity_test(g)
```

Maximum Matching Undirected Graph

```
#include <boost/graph/max_cardinality_matching.hpp>
vector<Vertex> mateMap(num_vertices(g));
bool success = checked_edmonds_maximum_cardinality_matching(g, &mateMap[0]);
int matching = matching_size(g, &mateMap[0]);

//Checking if vertex is unmatched:
mateMap[v_id] == graph_traits<Graph>::null_vertex()
```

Network Flow

```
#include <boost/tuple/tuple.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>

typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
  property<edge_capacity_t, long,
  property<edge_residual_capacity_t, long,
  property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;
typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;
typedef graph_traits<Graph>::edge_descriptor Edge;
```

```

void add_edge(int from, int to, int cap, Graph& g) {
    EdgeCapacityMap capacity = get(edge_capacity, g);
    ReverseEdgeMap reverse = get(edge_reverse, g);

    Edge there, back;
    tie(there, tuples::ignore) = add_edge(from, to, g);
    tie(back, tuples::ignore) = add_edge(to, from, g);
    capacity[there] = cap;
    capacity[back] = 0;
    reverse[there] = back;
    reverse[back] = there;
}
```

Max Flow

```
#include <boost/graph/push_relabel_max_flow.hpp>
int maxflow = push_relabel_max_flow(g, source, sink);
```

Vertex And Edge Types

```
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;
```

Undirected Graph with Edge properties

```
typedef adjacency_list<vecS, vecS, undirectedS, no_property, property<edge_index_t, int, ↙
    ↳ property<edge_weight_t, int> > > Graph;
```

Property Maps

```
typedef property_map<Graph, vertex_index_t>::type IndexMap;
typedef property_map<Graph, edge_weight_t>::type WeightMap;
typedef property_map<Graph, edge_index_t>::type EdgeIndexMap;
```

Instantiation Property Maps

```
EdgeIndexMap edgeIndex = get(edge_index, g);
WeightMap weightMap = get(edge_weight, g);
IndexMap indexMap = get(vertex_index, g);
```

Iterators

```
typedef graph_traits<Graph>::edge_iterator EI;
typedef graph_traits<Graph>::out_edge_iterator OEI;
```

Iterating over all edges

```
typedef graph_traits<Graph>::edge_iterator EI;
typedef graph_traits<Graph>::out_edge_iterator OEI;

EI ebegin, eend;
for(tie(ebegin, eend) = edges(g); ebegin != eend; ++ebegin) {
    // source(*ebegin, g);
    // target(*ebegin, g);
    // weightMap[*ebegin] = ...
}

OEI ebegin, eend;
for(tie(ebegin, eend) = out_edges(v, g); ebegin != eend; ++ebegin) {}
```

Iterating over all vertices

```
typedef graph_traits<Graph>::vertex_iterator VI;
for(tie(vbegin, vend) = vertices(g); vbegin != vend; ++vbegin) {
    // vbegin is of type vertex_descriptor
    // *vbegin is an integer
}
```

CGAL Cheatsheet

General

```
cin.sync_with_stdio(false);
cout << std::setiosflags(std::ios::fixed) << std::setprecision(0);
```

```
CGAL::has_smaller_distance_to_point(origin, p1, p2)    // true p1, false p2
CGAL::squared_distance(K::Point_2, K::Point_2);
```

Includes

```
#include <CGAL/Exact_predicates_exact_constructions_kernel_with_sqrt.h>
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
```

```
typedef CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt K;
typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
```

Ceil to double

```
double ceil_to_double(const K::FT& x) {
    double a = ceil(CGAL::to_double(x));
    while (a < x) a += 1;
    while (a-1 >= x) a -= 1;
    return a;
}
```

Floor to double

```
double floor_to_double(const K::FT& x) {
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}
```

Min Circle

```
#include <CGAL/Min_circle_2.h>
#include <CGAL/Min_circle_2_traits_2.h>
```

```
typedef CGAL::Min_circle_2_traits_2<K> Traits;
typedef CGAL::Min_circle_2<Traits> Min_circle;
```

```
Min_circle mc(points.begin(), points.end(), true);
for(Min_circle::Support_point_iterator it = mc.support_points_begin(); it != ↙
    ↳ mc.support_points_end(); ++it) {}
```

```
Traits::Circle c = mc.circle();
c.squared_radius();
```

Intersections

```
if(CGAL::do_intersect(ray, obstacle)) {    // could be any two geometric shapes.
    K::Point_2 intersection_point;
    CGAL::Object o = CGAL::intersection(ray, obstacle);
}
if(const K::Point_2* p = CGAL::object_cast<K::Point_2>(&o)) {
    intersection_point = *p;    // important: reference
}
else if (const K::Segment_2* s = CGAL::object_cast<K::Segment_2>(&o))
    intersection_point = s->source();    //important: now object ->.
}
else throw runtime_error("strange␣segment␣intersection");
```

Delauny Triangulation

```
#include <CGAL/Delaunay_triangulation_2.h>
typedef CGAL::Delaunay_triangulation_2<K> Delaunay;
// Find nearest Delaunay vertex:
Triangulation::Vertex_handle v = t.nearest_vertex(Point);
```

```
vector<K::Point_2> points;
Delaunay t;
t.insert(points.begin(), points.end());
```

Iterate over finite edges

```
typedef Delaunay::Finite_edges_iterator FEI;
```

```
for(FEI edge = t.finite_edges_begin(); edge != t.finite_edges_end(); ++edge) {
    Delaunay::Vertex_handle v1 = edge->first->vertex((edge->second + 1) % 3);
    Delaunay::Vertex_handle v2 = edge->first->vertex((edge->second + 2) % 3);
    K::Segment_2 seg = t.segment(edge);
}
```

Iterate over finite vertices

```
typedef Delaunay::Finite_vertices_iterator FVI;

for(FVI p = t.finite_vertices_begin(); p != t.finite_vertices_end(); ++p) {
    Delaunay::Vertex_handle vertex = p;
    vertex->point();    // returns K::Point_2
}
```

Vertex, Edge, Face information

Edge information in map

```
map<Delaunay::Vertex_handle, int> vertices;
map<Delaunay::Face_handle, int> faces;
map<Edge, int> edges;
```

Vertex base with info

```
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
typedef CGAL::Triangulation_vertex_base_with_info_2<int, K> Vb;
typedef CGAL::Triangulation_data_structure_2<Vb> Tds;
typedef CGAL::Delaunay_triangulation_2<K, Tds> Delaunay;
```

```
// set info while inserting
vector<pair<K::Point_2, int> > points;    // the second param is used as vertex id.
vector<K::Point_2> points;
Delaunay t;
t.insert(points.begin(), points.end());

// access info
Delaunay::Vertex_handle vertex = p;
p->info() = 1;
```

Face base with info

```
#include <CGAL/Triangulation_face_base_with_info_2.h>
Triangulation_face_base_with_info_2<int,Traits,Fb>
...
```

Linear Programming

```
#include <cassert>
#include <CGAL/basic.h>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
```

```
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>    // Use Gmpz for integers, Gmpq for double.
typedef CGAL::Gmpz ET;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float ET;
#endif
```

```
typedef CGAL::Quadratic_program<int> Program;
typedef CGAL::Quadratic_program_solution<ET> Solution;
```

```
Program qp(CGAL::SMALLER, true, 0, false, 0);

qp.set_a(COL j, ROW i, VAL);    // COL = variable, ROW = constraint
qp.set_b(ROW i, VAL);
qp.set_r(ROW i, CGAL::SMALLER/CGAL::LARGER);
qp.set_d(VAR1, VAR2, 2*VAL);    // for quadratic programs only
qp.set_c(VAR, VAL);
qp.set_c0(VAL);

Solution s = CGAL::solve_quadratic_program(qp, ET());
Solution s = CGAL::solve_linear_program(lp, ET());

s.is_optimal()
s.is_infeasible()
s.is_unbounded()
CGAL::to_double(s.objective_value())
```