------------- Chapter 2 ---------------
01-Array Vector

```cpp
#include <cstdio>
#include <vector>
using namespace std;

int main() {
  int arr[5] = {7,7,7};   // initial size (5) and initial value {7,7,7,0,0}
  vector<int> v(5, 5);    // initial size (5) and initial value {5,5,5,5,5}

  printf("arr[2] = %d and v[2] = %d\n", arr[2], v[2]);           // 7 and 5

  for (int i = 0; i < 5; i++) {
    arr[i] = i;
    v[i] = i;
  }

  printf("arr[2] = %d and v[2] = %d\n", arr[2], v[2]);           // 2 and 2

  // arr[5] = 5;     // static array will generate index out of bound error
  // uncomment the line above to see the error

  v.push_back(5);                              // but vector will resize itself
  printf("v[5] = %d\n", v[5]);                                          // 5

  return 0;
}
```

02-Algorithm Collections

```
#include <algorithm>
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

typedef struct {
  int id;
  int solved;
  int penalty;
} team;

bool icpc_cmp(team a, team b) {
  if (a.solved != b.solved) // can use this primary field to decide sorted order
    return a.solved > b.solved;   // ICPC rule: sort by number of problem solved
  else if (a.penalty != b.penalty)       // a.solved == b.solved, but we can use
                                       // secondary field to decide sorted order
    return a.penalty < b.penalty;       // ICPC rule: sort by descending penalty
  else                          // a.solved == b.solved AND a.penalty == b.penalty
    return a.id < b.id;                           // sort based on increasing team ID
}

int main() {
  int *pos, arr[] = {10, 7, 2, 15, 4};
  vector<int> v(arr, arr + 5);          // another way to initialize vector
  vector<int>::iterator j;

  // sort descending with vector
  sort(v.rbegin(), v.rend());        // example of using 'reverse iterator'
  for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    printf("%d ", *it);                        // access the value of iterator
  printf("\n");
  printf("==================\n");

  // sort descending with integer array
  sort(arr, arr + 5);                                               // ascending
  reverse(arr, arr + 5);                                         // then reverse
  for (int i = 0; i < 5; i++)
    printf("%d ", arr[i]);
  printf("\n");
  printf("==================\n");

  random_shuffle(v.begin(), v.end());          // shuffle the content again
  for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    printf("%d ", *it);
  printf("\n");
  printf("==================\n");
  partial_sort(v.begin(), v.begin() + 2, v.end());      // partial_sort demo
  for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    printf("%d ", *it);
  printf("\n");
  printf("==================\n");

  // sort ascending
  sort(arr, arr + 5);                              // arr should be sorted now
  for (int i = 0; i < 5; i++)                               // 2, 4, 7, 10, 15
    printf("%d ", arr[i]);
```

```
  printf("\n");
  sort(v.begin(), v.end());                // sorting a vector, same output
  for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    printf("%d ", *it);
  printf("\n");
  printf("==================\n");

  // multi-field sorting example, suppose we have 4 ICPC teams
  team nus[4] = { {1, 1, 10},
                  {2, 3, 60},
                  {3, 1, 20},
                  {4, 3, 60} };

  // without sorting, they will be ranked like this:
  for (int i = 0; i < 4; i++)
    printf("id: %d, solved: %d, penalty: %d\n",
           nus[i].id, nus[i].solved, nus[i].penalty);

  sort(nus, nus + 4, icpc_cmp);         // sort using a comparison function
  printf("==================\n");
  // after sorting using ICPC rule, they will be ranked like this:
  for (int i = 0; i < 4; i++)
    printf("id: %d, solved: %d, penalty: %d\n",
           nus[i].id, nus[i].solved, nus[i].penalty);
  printf("==================\n");

  // there is a trick for multi-field sorting if the sort order is "standard"
  // use "chained" pair class in C++ and put the highest priority in front
  typedef pair < int, pair < string, string > > state;
  state a = make_pair(10, make_pair("steven", "grace"));
  state b = make_pair(7, make_pair("steven", "halim"));
  state c = make_pair(7, make_pair("steven", "felix"));
  state d = make_pair(9, make_pair("a", "b"));
  vector<state> test;
  test.push_back(a);
  test.push_back(b);
  test.push_back(c);
  test.push_back(d);
  for (int i = 0; i < 4; i++)
    printf("value: %d, name1 = %s, name2 = %s\n", test[i].first,
    ((string)test[i].second.first).c_str(),
((string)test[i].second.second).c_str());
  printf("==================\n");
  sort(test.begin(), test.end());    // no need to use a comparison function
  // sorted ascending based on value, then based on name1,
  // then based on name2, in that order!
  for (int i = 0; i < 4; i++)
    printf("value: %d, name1 = %s, name2 = %s\n", test[i].first,
      ((string)test[i].second.first).c_str(),
((string)test[i].second.second).c_str());
  printf("==================\n");

  // binary search using lower bound
  pos = lower_bound(arr, arr + 5, 7);                              // found
  printf("%d\n", *pos);
  j = lower_bound(v.begin(), v.end(), 7);
  printf("%d\n", *j);

  pos = lower_bound(arr, arr + 5, 77);                         // not found
  if (pos - arr == 5) // arr is of size 5 ->
                      //    arr[0], arr[1], arr[2], arr[3], arr[4]
```

```
                      // if lower_bound cannot find the required value,
                      //   it will set return arr index +1 of arr size, i.e.
                      //   the 'non existent' arr[5]
                      // thus, testing whether pos - arr == 5 blocks
                      //   can detect this "not found" issue
    printf("77 not found\n");
  j = lower_bound(v.begin(), v.end(), 77);
  if (j == v.end()) // with vector, lower_bound will do the same:
                    //   return vector index +1 of vector size
                    // but this is exactly the position of vector.end()
                    //   so we can test "not found" this way
    printf("77 not found\n");
  printf("===================\n");

  // useful if you want to generate permutations of set
  next_permutation(arr, arr + 5); // 2, 4, 7, 10, 15 -> 2, 4, 7, 15, 10
  next_permutation(arr, arr + 5); // 2, 4, 7, 15, 10 -> 2, 4, 10, 7, 15
  for (int i = 0; i < 5; i++)
    printf("%d ", arr[i]);
  printf("\n");

  next_permutation(v.begin(), v.end());
  next_permutation(v.begin(), v.end());
  for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    printf("%d ", *it);
  printf("\n");
  printf("===================\n");

  // sometimes these two useful simple macros are used
  printf("min(10, 7) = %d\n", min(10, 7));
  printf("max(10, 7) = %d\n", max(10, 7));

  return 0;
}
```

03-bit manipulation

```
// note: for example usage of bitset, see ch5_06_primes.cpp

#include <cmath>
#include <cstdio>
#include <stack>
using namespace std;

#define isOn(S, j) (S & (1 << j))
#define setBit(S, j) (S |= (1 << j))
#define clearBit(S, j) (S &= ~(1 << j))
#define toggleBit(S, j) (S ^= (1 << j))
#define lowBit(S) (S & (-S))
#define setAll(S, n) (S = (1 << n) - 1)

#define modulo(S, N) ((S) & (N - 1))   // returns S % N, where N is a power of 2
#define isPowerOfTwo(S) (!(S & (S - 1)))
#define nearestPowerOfTwo(S) ((int)pow(2.0, (int)((log((double)S) / log(2.0)) +
0.5)))
#define turnOffLastBit(S) ((S) & (S - 1))
#define turnOnLastZero(S) ((S) | (S + 1))
#define turnOffLastConsecutiveBits(S) ((S) & (S + 1))
#define turnOnLastConsecutiveZeroes(S) ((S) | (S - 1))

void printSet(int vS) {                         // in binary representation
  printf("S = %2d = ", vS);
  stack<int> st;
  while (vS)
    st.push(vS % 2), vS /= 2;
  while (!st.empty())                         // to reverse the print order
    printf("%d", st.top()), st.pop();
  printf("\n");
}

int main() {
  int S, T;

  printf("1. Representation (all indexing are 0-based and counted from
right)\n");
  S = 34; printSet(S);
  printf("\n");

  printf("2. Multiply S by 2, then divide S by 4 (2x2), then by 2\n");
  S = 34; printSet(S);
  S = S << 1; printSet(S);
  S = S >> 2; printSet(S);
  S = S >> 1; printSet(S);
  printf("\n");

  printf("3. Set/turn on the 3-th item of the set\n");
  S = 34; printSet(S);
  setBit(S, 3); printSet(S);
  printf("\n");

  printf("4. Check if the 3-th and then 2-nd item of the set is on?\n");
  S = 42; printSet(S);
  T = isOn(S, 3); printf("T = %d, %s\n", T, T ? "ON" : "OFF");
  T = isOn(S, 2); printf("T = %d, %s\n", T, T ? "ON" : "OFF");
```

```
  printf("\n");

  printf("5. Clear/turn off the 1-st item of the set\n");
  S = 42; printSet(S);
  clearBit(S, 1); printSet(S);
  printf("\n");

  printf("6. Toggle the 2-nd item and then 3-rd item of the set\n");
  S = 40; printSet(S);
  toggleBit(S, 2); printSet(S);
  toggleBit(S, 3); printSet(S);
  printf("\n");

  printf("7. Check the first bit from right that is on\n");
  S = 40; printSet(S);
  T = lowBit(S); printf("T = %d (this is always a power of 2)\n", T);
  S = 52; printSet(S);
  T = lowBit(S); printf("T = %d (this is always a power of 2)\n", T);
  printf("\n");

  printf("8. Turn on all bits in a set of size n = 6\n");
  setAll(S, 6); printSet(S);
  printf("\n");

  printf("9. Other tricks (not shown in the book)\n");
  printf("8 %c 4 = %d\n", '%', modulo(8, 4));
  printf("7 %c 4 = %d\n", '%', modulo(7, 4));
  printf("6 %c 4 = %d\n", '%', modulo(6, 4));
  printf("5 %c 4 = %d\n", '%', modulo(5, 4));
  printf("is %d power of two? %d\n", 9, isPowerOfTwo(9));
  printf("is %d power of two? %d\n", 8, isPowerOfTwo(8));
  printf("is %d power of two? %d\n", 7, isPowerOfTwo(7));
  for (int i = 0; i <= 16; i++)
    printf("Nearest power of two of %d is %d\n", i, nearestPowerOfTwo(i));
  printf("S = %d, turn off last bit in S, S = %d\n", 40, turnOffLastBit(40));
  printf("S = %d, turn on last zero in S, S = %d\n", 41, turnOnLastZero(41));
  printf("S = %d, turn off last consectuve bits in S, S = %d\n", 39,
turnOffLastConsecutiveBits(39));
  printf("S = %d, turn on last consecutive zeroes in S, S = %d\n", 36,
turnOnLastConsecutiveZeroes(36));

  return 0;
}
```

04-Stack queue

```cpp
#include <cstdio>
#include <stack>
#include <queue>
using namespace std;

int main() {
  stack<char> s;
  queue<char> q;
  deque<char> d;

  printf("%d\n", s.empty());                   // currently s is empty, true (1)
  printf("==================\n");
  s.push('a');
  s.push('b');
  s.push('c');
  // stack is LIFO, thus the content of s is currently like this:
  // c <- top
  // b
  // a
  printf("%c\n", s.top());                                      // output 'c'
  s.pop();                                                      // pop topmost
  printf("%c\n", s.top());                                      // output 'b'
  printf("%d\n", s.empty());          // currently s is not empty, false (0)
  printf("==================\n");

  printf("%d\n", q.empty());               // currently q is empty, true (1)
  printf("==================\n");
  while (!s.empty()) {                     // stack s still has 2 more items
    q.push(s.top());                            // enqueue 'b', and then 'a'
    s.pop();
  }
  q.push('z');                                         // add one more item
  printf("%c\n", q.front());                                    // prints 'b'
  printf("%c\n", q.back());                                     // prints 'z'

  // output 'b', 'a', then 'z' (until queue is empty), according to the
insertion order above
  printf("==================\n");
  while (!q.empty()) {
    printf("%c\n", q.front());                       // take the front first
    q.pop();                               // before popping (dequeue-ing) it
  }

  printf("==================\n");
  d.push_back('a');
  d.push_back('b');
  d.push_back('c');
  printf("%c - %c\n", d.front(), d.back());               // prints 'a - c'
  d.push_front('d');
  printf("%c - %c\n", d.front(), d.back());               // prints 'd - c'
  d.pop_back();
  printf("%c - %c\n", d.front(), d.back());               // prints 'd - b'
  d.pop_front();
  printf("%c - %c\n", d.front(), d.back());               // prints 'a - b'

  return 0;
}
```

05-map set

```
#include <cstdio>
#include <map>
#include <set>
#include <string>
using namespace std;

int main() {
  char name[20];
  int value;
  // note: there are many clever usages of this set/map
  // that you can learn by looking at top coder's codes
  // note, we don't have to use .clear() if we have just initialized the set/map
  set<int> used_values; // used_values.clear();
  map<string, int> mapper; // mapper.clear();

  // suppose we enter these 7 name-score pairs below
  /*
  john 78
  billy 69
  andy 80
  steven 77
  felix 82
  grace 75
  martin 81
  */
  mapper["john"] = 78;   used_values.insert(78);
  mapper["billy"] = 69;  used_values.insert(69);
  mapper["andy"] = 80;   used_values.insert(80);
  mapper["steven"] = 77; used_values.insert(77);
  mapper["felix"] = 82;  used_values.insert(82);
  mapper["grace"] = 75;  used_values.insert(75);
  mapper["martin"] = 81; used_values.insert(81);

  // then the internal content of mapper MAY be something like this:
  // re-read balanced BST concept if you do not understand this diagram
  // the keys are names (string)!
  //                        (grace,75)
  //           (billy,69)               (martin,81)
  //     (andy,80)   (felix,82)    (john,78)  (steven,77)

  // iterating through the content of mapper will give a sorted output
  // based on keys (names)
  for (map<string, int>::iterator it = mapper.begin(); it != mapper.end(); it++)
    printf("%s %d\n", ((string)it->first).c_str(), it->second);

  // map can also be used like this
  printf("steven's score is %d, grace's score is %d\n",
    mapper["steven"], mapper["grace"]);
  printf("==================\n");

  // interesting usage of lower_bound and upper_bound
  // display data between ["f".."m") ('felix' is included, martin' is excluded)
  for (map<string, int>::iterator it = mapper.lower_bound("f"); it !=
mapper.upper_bound("m"); it++)
    printf("%s %d\n", ((string)it->first).c_str(), it->second);

  // the internal content of used_values MAY be something like this
```

```
  // the keys are values (integers)!
  //                 (78)
  //         (75)            (81)
  //     (69)    (77)    (80)    (82)

  // O(log n) search, found
  printf("%d\n", *used_values.find(77));
  // returns [69, 75] (these two are before 77 in the inorder traversal of this
BST)
  for (set<int>::iterator it = used_values.begin(); it !=
used_values.lower_bound(77); it++)
    printf("%d,", *it);
  printf("\n");
  // returns [77, 78, 80, 81, 82] (these five are equal or after 77 in the
inorder traversal of this BST)
  for (set<int>::iterator it = used_values.lower_bound(77); it !=
used_values.end(); it++)
    printf("%d,", *it);
  printf("\n");
  // O(log n) search, not found
  if (used_values.find(79) == used_values.end())
    printf("79 not found\n");

  return 0;
}
```

06-priority queue

```
#include <cstdio>
#include <iostream>
#include <string>
#include <queue>
using namespace std;

int main() {
  int money;
  char name[20];
  priority_queue< pair<int, string> > pq;             // introducing 'pair'
  pair<int, string> result;

  // suppose we enter these 7 money-name pairs below
  /*
  100 john
  10 billy
  20 andy
  100 steven
  70 felix
  2000 grace
  70 martin
  */
  pq.push(make_pair(100, "john"));          // inserting a pair in O(log n)
  pq.push(make_pair(10, "billy"));
  pq.push(make_pair(20, "andy"));
  pq.push(make_pair(100, "steven"));
  pq.push(make_pair(70, "felix"));
  pq.push(make_pair(2000, "grace"));
  pq.push(make_pair(70, "martin"));
  // priority queue will arrange items in 'heap' based
  // on the first key in pair, which is money (integer), largest first
  // if first keys tie, use second key, which is name, largest first

  // the internal content of pq heap MAY be something like this:
  // re-read (max) heap concept if you do not understand this diagram
  // the primary keys are money (integer), secondary keys are names (string)!
  //                        (2000,grace)
  //           (100,steven)               (70,martin)
  //     (100,john)   (10,billy)     (20,andy)  (70,felix)

  // let's print out the top 3 person with most money
  result = pq.top();                 // O(1) to access the top / max element
  pq.pop();          // O(log n) to delete the top and repair the structure
  printf("%s has %d $\n", ((string)result.second).c_str(), result.first);
  result = pq.top(); pq.pop();
  printf("%s has %d $\n", ((string)result.second).c_str(), result.first);
  result = pq.top(); pq.pop();
  printf("%s has %d $\n", ((string)result.second).c_str(), result.first);

  return 0;
}
```

07-graph ds

```
#include <cstdio>
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

typedef pair<int, int> ii;
typedef vector<ii> vii;

int main() {
  int V, E, total_neighbors, id, weight, a, b;
  int AdjMat[100][100];
  vector<vii> AdjList;
  priority_queue< pair<int, ii> > EdgeList;   // one way to store Edge List

  // Try this input for Adjacency Matrix/List/EdgeList
  // Adj Matrix
  //   for each line: |V| entries, 0 or the weight
  // Adj List
  //   for each line: num neighbors, list of neighbors + weight pairs
  // Edge List
  //   for each line: a-b of edge(a,b) and weight
  /*
  6
    0  10   0   0 100   0
   10   0   7   0   8   0
    0   7   0   9   0   0
    0   0   9   0  20   5
  100   8   0  20   0   0
    0   0   0   5   0   0
  6
  2 2 10 5 100
  3 1 10 3 7 5 8
  2 2 7 4 9
  3 3 9 5 20 6 5
  3 1 100 2 8 4 20
  1 4 5
  7
  1 2 10
  1 5 100
  2 3 7
  2 5 8
  3 4 9
  4 5 20
  4 6 5
  */
  freopen("in_07.txt", "r", stdin);

  scanf("%d", &V);                               // we must know this size first!
                       // remember that if V is > 100, try NOT to use AdjMat!
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      scanf("%d", &AdjMat[i][j]);

  printf("Neighbors of vertex 0:\n");
  for (int j = 0; j < V; j++)                                            // O(|V|)
    if (AdjMat[0][j])
```

```
      printf("Edge 0-%d (weight = %d)\n", j, AdjMat[0][j]);

  scanf("%d", &V);
  AdjList.assign(V, vii()); // quick way to initialize AdjList with V entries of
vii
  for (int i = 0; i < V; i++) {
    scanf("%d", &total_neighbors);
    for (int j = 0; j < total_neighbors; j++) {
      scanf("%d %d", &id, &weight);
      AdjList[i].push_back(ii(id - 1, weight));    // some index adjustment
    }
  }

  printf("Neighbors of vertex 0:\n");
  for (vii::iterator j = AdjList[0].begin(); j != AdjList[0].end(); j++)
    // AdjList[0] contains the required information
    // O(k), where k is the number of neighbors
    printf("Edge 0-%d (weight = %d)\n", j->first, j->second);

  scanf("%d", &E);
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &a, &b, &weight);
    EdgeList.push(make_pair(-weight, ii(a, b))); // trick to reverse sort order
  }

  // edges sorted by weight (smallest->largest)
  for (int i = 0; i < E; i++) {
    pair<int, ii> edge = EdgeList.top(); EdgeList.pop();
    // negate the weight again
    printf("weight: %d (%d-%d)\n", -edge.first, edge.second.first,
edge.second.second);
  }

  return 0;
}
```

08-union find ds

```
#include <cstdio>
#include <vector>
using namespace std;

typedef vector<int> vi;

// Union-Find Disjoint Sets Library written in OOP manner, using both path
compression and union by rank heuristics
class UnionFind {                                              // OOP style
private:
  vi p, rank, setSize;                           // remember: vi is vector<int>
  int numSets;
public:
  UnionFind(int N) {
    setSize.assign(N, 1); numSets = N; rank.assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) { numSets--;
    int x = findSet(i), y = findSet(j);
    // rank is used to keep the tree short
    if (rank[x] > rank[y]) { p[y] = x; setSize[x] += setSize[y]; }
    else                   { p[x] = y; setSize[y] += setSize[x];
                             if (rank[x] == rank[y]) rank[y]++; } } }
  int numDisjointSets() { return numSets; }
  int sizeOfSet(int i) { return setSize[findSet(i)]; }
};

int main() {
  printf("Assume that there are 5 disjoint sets initially\n");
  UnionFind UF(5); // create 5 disjoint sets
  printf("%d\n", UF.numDisjointSets()); // 5
  UF.unionSet(0, 1);
  printf("%d\n", UF.numDisjointSets()); // 4
  UF.unionSet(2, 3);
  printf("%d\n", UF.numDisjointSets()); // 3
  UF.unionSet(4, 3);
  printf("%d\n", UF.numDisjointSets()); // 2
  printf("isSameSet(0, 3) = %d\n", UF.isSameSet(0, 3)); // will return 0 (false)
  printf("isSameSet(4, 3) = %d\n", UF.isSameSet(4, 3)); // will return 1 (true)
  for (int i = 0; i < 5; i++) // findSet will return 1 for {0, 1} and 3 for {2,
3, 4}
    printf("findSet(%d) = %d, sizeOfSet(%d) = %d\n", i, UF.findSet(i), i,
UF.sizeOfSet(i));
  UF.unionSet(0, 3);
  printf("%d\n", UF.numDisjointSets()); // 1
  for (int i = 0; i < 5; i++) // findSet will return 3 for {0, 1, 2, 3, 4}
    printf("findSet(%d) = %d, sizeOfSet(%d) = %d\n", i, UF.findSet(i), i,
UF.sizeOfSet(i));
  return 0;
}
```

09-Segment Tree ds

```
#include <cmath>
#include <cstdio>
#include <vector>
using namespace std;

typedef vector<int> vi;

class SegmentTree {          // the segment tree is stored like a heap array
private: vi st, A;             // recall that vi is: typedef vector<int> vi;
  int n;
  int left (int p) { return p << 1; }     // same as binary heap operations
  int right(int p) { return (p << 1) + 1; }

  void build(int p, int L, int R) {                            // O(n log n)
    if (L == R)                                 // as L == R, either one is fine
      st[p] = L;                                             // store the index
    else {                                      // recursively compute the values
      build(left(p) , L              , (L + R) / 2);
      build(right(p), (L + R) / 2 + 1, R          );
      int p1 = st[left(p)], p2 = st[right(p)];
      st[p] = (A[p1] <= A[p2]) ? p1 : p2;
  } }

  int rmq(int p, int L, int R, int i, int j) {                   // O(log n)
    if (i >  R || j <  L) return -1; // current segment outside query range
    if (L >= i && R <= j) return st[p];                 // inside query range

     // compute the min position in the left and right part of the interval
    int p1 = rmq(left(p) , L            , (L+R) / 2, i, j);
    int p2 = rmq(right(p), (L+R) / 2 + 1, R          , i, j);

    if (p1 == -1) return p2;   // if we try to access segment outside query
    if (p2 == -1) return p1;                              // same as above
    return (A[p1] <= A[p2]) ? p1 : p2; }            // as as in build routine

  int update_point(int p, int L, int R, int idx, int new_value) {
    // this update code is still preliminary, i == j
    // must be able to update range in the future!
    int i = idx, j = idx;

    // if the current interval does not intersect
    // the update interval, return this st node value!
    if (i > R || j < L)
      return st[p];

    // if the current interval is included in the update range,
    // update that st[node]
    if (L == i && R == j) {
      A[i] = new_value; // update the underlying array
      return st[p] = L; // this index
    }

    // compute the minimum pition in the
    // left and right part of the interval
    int p1, p2;
    p1 = update_point(left(p) , L              , (L + R) / 2, idx, new_value);
    p2 = update_point(right(p), (L + R) / 2 + 1, R          , idx, new_value);
```

```
    // return the pition where the overall minimum is
    return st[p] = (A[p1] <= A[p2]) ? p1 : p2;
  }

public:
  SegmentTree(const vi &_A) {
    A = _A; n = (int)A.size();              // copy content for local usage
    st.assign(4 * n, 0);            // create large enough vector of zeroes
    build(1, 0, n - 1);                                  // recursive build
  }

  int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); }   // overloading

  int update_point(int idx, int new_value) {
    return update_point(1, 0, n - 1, idx, new_value); }
};

int main() {
  int arr[] = { 18, 17, 13, 19, 15, 11, 20 };         // the original array
  vi A(arr, arr + 7);                          // copy the contents to a vector
  SegmentTree st(A);

  printf("              idx    0, 1, 2, 3, 4,  5, 6\n");
  printf("              A is {18,17,13,19,15, 11,20}\n");
  printf("RMQ(1, 3) = %d\n", st.rmq(1, 3));             // answer = index 2
  printf("RMQ(4, 6) = %d\n", st.rmq(4, 6));             // answer = index 5
  printf("RMQ(3, 4) = %d\n", st.rmq(3, 4));             // answer = index 4
  printf("RMQ(0, 0) = %d\n", st.rmq(0, 0));             // answer = index 0
  printf("RMQ(0, 1) = %d\n", st.rmq(0, 1));             // answer = index 1
  printf("RMQ(0, 6) = %d\n", st.rmq(0, 6));             // answer = index 5

  printf("              idx    0, 1, 2, 3, 4,  5, 6\n");
  printf("Now, modify A into {18,17,13,19,15,100,20}\n");
  st.update_point(5, 100);                    // update A[5] from 11 to 100
  printf("These values do not change\n");
  printf("RMQ(1, 3) = %d\n", st.rmq(1, 3));                            // 2
  printf("RMQ(3, 4) = %d\n", st.rmq(3, 4));                            // 4
  printf("RMQ(0, 0) = %d\n", st.rmq(0, 0));                            // 0
  printf("RMQ(0, 1) = %d\n", st.rmq(0, 1));                            // 1
  printf("These values change\n");
  printf("RMQ(0, 6) = %d\n", st.rmq(0, 6));                         // 5->2
  printf("RMQ(4, 6) = %d\n", st.rmq(4, 6));                         // 5->4
  printf("RMQ(4, 5) = %d\n", st.rmq(4, 5));                         // 5->4

  return 0;
}
```

10-Fenwick Tree ds

```
#include <cstdio>
#include <vector>
using namespace std;

typedef vector<int> vi;
#define LSOne(S) (S & (-S))

class FenwickTree {
private:
  vi ft;

public:
  FenwickTree() {}
  // initialization: n + 1 zeroes, ignore index 0
  FenwickTree(int n) { ft.assign(n + 1, 0); }

  int rsq(int b) {                                     // returns RSQ(1, b)
    int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
    return sum; }

  int rsq(int a, int b) {                              // returns RSQ(a, b)
    return rsq(b) - (a == 1 ? 0 : rsq(a - 1)); }

  // adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
  void adjust(int k, int v) {                    // note: n = ft.size() - 1
    for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }
};

int main() {                  // idx   0 1 2 3 4 5 6 7  8 9 10, no index 0!
  FenwickTree ft(10);         // ft = {-,0,0,0,0,0,0,0, 0,0,0}
  ft.adjust(2, 1);            // ft = {-,0,1,0,1,0,0,0, 1,0,0}, idx 2,4,8 => +1
  ft.adjust(4, 1);            // ft = {-,0,1,0,2,0,0,0, 2,0,0}, idx 4,8 => +1
  ft.adjust(5, 2);            // ft = {-,0,1,0,2,2,2,0, 4,0,0}, idx 5,6,8 => +2
  ft.adjust(6, 3);            // ft = {-,0,1,0,2,2,5,0, 7,0,0}, idx 6,8 => +3
  ft.adjust(7, 2);            // ft = {-,0,1,0,2,2,5,2, 9,0,0}, idx 7,8 => +2
  ft.adjust(8, 1);            // ft = {-,0,1,0,2,2,5,2,10,0,0}, idx 8 => +1
  ft.adjust(9, 1);            // ft = {-,0,1,0,2,2,5,2,10,1,1}, idx 9,10 => +1
  printf("%d\n", ft.rsq(1, 1));  // 0 => ft[1] = 0
  printf("%d\n", ft.rsq(1, 2));  // 1 => ft[2] = 1
  printf("%d\n", ft.rsq(1, 6));  // 7 => ft[6] + ft[4] = 5 + 2 = 7
  printf("%d\n", ft.rsq(1, 10)); // 11 => ft[10] + ft[8] = 1 + 10 = 11
  printf("%d\n", ft.rsq(3, 6));  // 6 => rsq(1, 6) - rsq(1, 2) = 7 - 1

  ft.adjust(5, 2); // update demo
  printf("%d\n", ft.rsq(1, 10)); // now 13
} // return 0;
```

------------- Chapter 4 ---------------
01-DFS

```
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

typedef pair<int, int> ii;      // In this chapter, we will frequently use these
typedef vector<ii> vii;      // three data type shortcuts. They may look cryptic
typedef vector<int> vi;   // but shortcuts are useful in competitive programming

#define DFS_WHITE -1 // normal DFS, do not change this with other values (other than 0), because we usually use memset with conjunction with DFS_WHITE
#define DFS_BLACK 1

vector<vii> AdjList;

void printThis(char* message) {
  printf("==================================\n");
  printf("%s\n", message);
  printf("==================================\n");
}

vi dfs_num;     // this variable has to be global, we cannot put it in recursion
int numCC;

void dfs(int u) {          // DFS for normal usage: as graph traversal algorithm
  printf(" %d", u);                                    // this vertex is visited
  dfs_num[u] = DFS_BLACK;      // important step: we mark this vertex as visited
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];                      // v is a (neighbor, weight) pair
    if (dfs_num[v.first] == DFS_WHITE)         // important check to avoid cycle
      dfs(v.first);      // recursively visits unvisited neighbors v of vertex u
} }

// note: this is not the version on implicit graph
void floodfill(int u, int color) {
  dfs_num[u] = color;                            // not just a generic DFS_BLACK
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      floodfill(v.first, color);
} }

vi topoSort;             // global vector to store the toposort in reverse order

void dfs2(int u) {    // change function name to differentiate with original dfs
  dfs_num[u] = DFS_BLACK;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      dfs2(v.first);
  }
  topoSort.push_back(u); }                        // that is, this is the only change

#define DFS_GRAY 2              // one more color for graph edges property check
vi dfs_parent;      // to differentiate real back edge versus bidirectional edge
```

```
void graphCheck(int u) {               // DFS for checking graph edge properties
  dfs_num[u] = DFS_GRAY;   // color this as DFS_GRAY (temp) instead of DFS_BLACK
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE) {     // Tree Edge, DFS_GRAY to DFS_WHITE
      dfs_parent[v.first] = u;                  // parent of this children is me
      graphCheck(v.first);
    }
    else if (dfs_num[v.first] == DFS_GRAY) {             // DFS_GRAY to DFS_GRAY
      if (v.first == dfs_parent[u])          // to differentiate these two cases
        printf(" Bidirectional (%d, %d) - (%d, %d)\n", u, v.first, v.first, u);
      else  // the most frequent application: check if the given graph is cyclic
        printf(" Back Edge (%d, %d) (Cycle)\n", u, v.first);
    }
    else if (dfs_num[v.first] == DFS_BLACK)             // DFS_GRAY to DFS_BLACK
      printf(" Forward/Cross Edge (%d, %d)\n", u, v.first);
  }
  dfs_num[u] = DFS_BLACK;     // after recursion, color this as DFS_BLACK (DONE)
}

vi dfs_low;       // additional information for articulation points/bridges/SCCs
vi articulation_vertex;
int dfsNumberCounter, dfsRoot, rootChildren;

void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;      // dfs_low[u] <= dfs_num[u]
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE) {                          // a tree edge
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;  // special case, count children of root

      articulationPointAndBridge(v.first);

      if (dfs_low[v.first] >= dfs_num[u])              // for articulation point
        articulation_vertex[u] = true;           // store this information first
      if (dfs_low[v.first] > dfs_num[u])                           // for bridge
        printf(" Edge (%d, %d) is a bridge\n", u, v.first);
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);       // update dfs_low[u]
    }
    else if (v.first != dfs_parent[u])       // a back edge and not direct cycle
      dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);       // update dfs_low[u]
} }

vi S, visited;                                    // additional global variables
int numSCC;

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;      // dfs_low[u] <= dfs_num[u]
  S.push_back(u);           // stores u in a vector based on order of visitation
  visited[u] = 1;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      tarjanSCC(v.first);
    if (visited[v.first])                                 // condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
  }

  if (dfs_low[u] == dfs_num[u]) {         // if this is a root (start) of an SCC
    printf("SCC %d:", ++numSCC);            // this part is done after recursion
```

```
    while (1) {
      int v = S.back(); S.pop_back(); visited[v] = 0;
      printf(" %d", v);
      if (u == v) break;
    }
    printf("\n");
} }

int main() {
  int V, total_neighbors, id, weight;

  /*
  // Use the following input:
  // Graph in Figure 4.1
  9
  1 1 0
  3 0 0 2 0 3 0
  2 1 0 3 0
  3 1 0 2 0 4 0
  1 3 0
  0
  2 7 0 8 0
  1 6 0
  1 6 0

  // Example of directed acyclic graph in Figure 4.4 (for toposort)
  8
  2 1 0 2 0
  2 2 0 3 0
  2 3 0 5 0
  1 4 0
  0
  0
  0
  1 6 0

  // Example of directed graph with back edges
  3
  1 1 0
  1 2 0
  1 0 0

  // Left graph in Figure 4.6/4.7/4.8
  6
  1 1 0
  3 0 0 2 0 4 0
  1 1 0
  1 4 0
  3 1 0 3 0 5 0
  1 4 0

  // Right graph in Figure 4.6/4.7/4.8
  6
  1 1 0
  5 0 0 2 0 3 0 4 0 5 0
  1 1 0
  1 1 0
  2 1 0 5 0
  2 1 0 4 0

  // Directed graph in Figure 4.9
```

```
8
1 1 0
1 3 0
1 1 0
2 2 0 4 0
1 5 0
1 7 0
1 4 0
1 6 0
*/

freopen("in_01.txt", "r", stdin);

scanf("%d", &V);
AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to
AdjList
for (int i = 0; i < V; i++) {
  scanf("%d", &total_neighbors);
  for (int j = 0; j < total_neighbors; j++) {
    scanf("%d %d", &id, &weight);
    AdjList[i].push_back(ii(id, weight));
  }
}

printThis("Standard DFS Demo (the input graph must be UNDIRECTED)");
numCC = 0;
dfs_num.assign(V, DFS_WHITE);    // this sets all vertices' state to DFS_WHITE
for (int i = 0; i < V; i++)                     // for each vertex i in [0..V-1]
  if (dfs_num[i] == DFS_WHITE)             // if that vertex is not visited yet
    printf("Component %d:", ++numCC), dfs(i), printf("\n");   // 3 lines here!
printf("There are %d connected components\n", numCC);

printThis("Flood Fill Demo (the input graph must be UNDIRECTED)");
numCC = 0;
dfs_num.assign(V, DFS_WHITE);
for (int i = 0; i < V; i++)
  if (dfs_num[i] == DFS_WHITE)
    floodfill(i, ++numCC);
for (int i = 0; i < V; i++)
  printf("Vertex %d has color %d\n", i, dfs_num[i]);

// make sure that the given graph is DAG
printThis("Topological Sort (the input graph must be DAG)");
topoSort.clear();
dfs_num.assign(V, DFS_WHITE);
for (int i = 0; i < V; i++)            // this part is the same as finding CCs
  if (dfs_num[i] == DFS_WHITE)
    dfs2(i);
reverse(topoSort.begin(), topoSort.end());                 // reverse topoSort
for (int i = 0; i < (int)topoSort.size(); i++)       // or you can simply read
  printf(" %d", topoSort[i]);           // the content of `topoSort' backwards
printf("\n");

printThis("Graph Edges Property Check");
numCC = 0;
dfs_num.assign(V, DFS_WHITE); dfs_parent.assign(V, -1);
for (int i = 0; i < V; i++)
  if (dfs_num[i] == DFS_WHITE)
    printf("Component %d:\n", ++numCC), graphCheck(i);       // 2 lines in one
```

```
  printThis("Articulation Points & Bridges (the input graph must be 
UNDIRECTED)");
  dfsNumberCounter = 0; dfs_num.assign(V, DFS_WHITE); dfs_low.assign(V, 0);
  dfs_parent.assign(V, -1); articulation_vertex.assign(V, 0);
  printf("Bridges:\n");
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE) {
      dfsRoot = i; rootChildren = 0;
      articulationPointAndBridge(i);
      articulation_vertex[dfsRoot] = (rootChildren > 1); }       // special case
  printf("Articulation Points:\n");
  for (int i = 0; i < V; i++)
    if (articulation_vertex[i])
      printf(" Vertex %d\n", i);

  printThis("Strongly Connected Components (the input graph must be DIRECTED)");
  dfs_num.assign(V, DFS_WHITE); dfs_low.assign(V, 0); visited.assign(V, 0);
  dfsNumberCounter = numSCC = 0;
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE)
      tarjanSCC(i);

  return 0;
}
/* Wetlands of Florida */
```

02-classic DFS flood fill

```
#include <cstdio>
#include <cstring>
using namespace std;

#define REP(i, a, b) \
  for (int i = int(a); i <= int(b); i++)

char line[150], grid[150][150];
int TC, R, C, row, col;

int dr[] = {1,1,0,-1,-1,-1, 0, 1}; // S,SE,E,NE,N,NW,W,SW
int dc[] = {0,1,1, 1, 0,-1,-1,-1}; // neighbors
int floodfill(int r, int c, char c1, char c2) {
  if (r<0 || r>=R || c<0 || c>=C) return 0; // outside
  if (grid[r][c] != c1) return 0; // we want only c1
  grid[r][c] = c2; // important step to avoid cycling!
  int ans = 1; // coloring c1 -> c2, add 1 to answer
  REP (d, 0, 7) // recurse to neighbors
    ans += floodfill(r + dr[d], c + dc[d], c1, c2);
  return ans;
}

// inside the int main() of the solution for UVa 469 - Wetlands of Florida
int main() {
  // read the implicit graph as global 2D array 'grid'/R/C and (row, col) query
coordinate
  sscanf(gets(line), "%d", &TC);
  gets(line); // remove dummy line

  while (TC--) {
    R = 0;
    while (1) {
      gets(grid[R]);
      if (grid[R][0] != 'L' && grid[R][0] != 'W') // start of query
        break;
      R++;
    }
    C = (int)strlen(grid[0]);

    strcpy(line, grid[R]);
    while (1) {
      sscanf(line, "%d %d", &row, &col); row--; col--; // index starts from 0!
      printf("%d\n", floodfill(row, col, 'W', '.')); // change water 'W' to '.';
count size of this lake
      floodfill(row, col, '.', 'W'); // restore for next query
      gets(line);
      if (strcmp(line, "") == 0 || feof(stdin)) // next test case or last test
case
        break;
    }

    if (TC)
      printf("\n");
  }

  return 0;
```

```
}
```

03-Kruskal Prim

```
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;

// Union-Find Disjoint Sets Library written in OOP manner, using both path
compression and union by rank heuristics
class UnionFind {                                              // OOP style
private:
  vi p, rank, setSize;                       // remember: vi is vector<int>
  int numSets;
public:
  UnionFind(int N) {
    setSize.assign(N, 1); numSets = N; rank.assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) { numSets--;
    int x = findSet(i), y = findSet(j);
    // rank is used to keep the tree short
    if (rank[x] > rank[y]) { p[y] = x; setSize[x] += setSize[y]; }
    else                   { p[x] = y; setSize[y] += setSize[x];
                             if (rank[x] == rank[y]) rank[y]++; } } }
  int numDisjointSets() { return numSets; }
  int sizeOfSet(int i) { return setSize[findSet(i)]; }
};

vector<vii> AdjList;
vi taken;                                  // global boolean flag to avoid cycle
priority_queue<ii> pq;            // priority queue to help choose shorter edges

void process(int vtx) {    // so, we use -ve sign to reverse the sort order
  taken[vtx] = 1;
  for (int j = 0; j < (int)AdjList[vtx].size(); j++) {
    ii v = AdjList[vtx][j];
    if (!taken[v.first]) pq.push(ii(-v.second, -v.first));
} }                                 // sort by (inc) weight then by (inc) id

int main() {
  int V, E, u, v, w;

  /*
  // Graph in Figure 4.10 left, format: list of weighted edges
  // This example shows another form of reading graph input
  5 7
  0 1 4
  0 2 4
  0 3 6
  0 4 6
```

```
  1 2 2
  2 3 8
  3 4 9
  */

  freopen("in_03.txt", "r", stdin);

  scanf("%d %d", &V, &E);
  // Kruskal's algorithm merged with Prim's algorithm
  AdjList.assign(V, vii());
  vector< pair<int, ii> > EdgeList;   // (weight, two vertices) of the edge
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);            // read the triple: (u, v, w)
    EdgeList.push_back(make_pair(w, ii(u, v)));                // (w, u, v)
    AdjList[u].push_back(ii(v, w));
    AdjList[v].push_back(ii(u, w));
  }
  sort(EdgeList.begin(), EdgeList.end()); // sort by edge weight O(E log E)
                      // note: pair object has built-in comparison function

  int mst_cost = 0;
  UnionFind UF(V);                     // all V are disjoint sets initially
  for (int i = 0; i < E; i++) {                      // for each edge, O(E)
    pair<int, ii> front = EdgeList[i];
    if (!UF.isSameSet(front.second.first, front.second.second)) {  // check
      mst_cost += front.first;                 // add the weight of e to MST
      UF.unionSet(front.second.first, front.second.second);    // link them
  } }                         // note: the runtime cost of UFDS is very light

  // note: the number of disjoint sets must eventually be 1 for a valid MST
  printf("MST cost = %d (Kruskal's)\n", mst_cost);



// inside int main() --- assume the graph is stored in AdjList, pq is empty
  taken.assign(V, 0);                  // no vertex is taken at the beginning
  process(0);   // take vertex 0 and process all edges incident to vertex 0
  mst_cost = 0;
  while (!pq.empty()) {  // repeat until V vertices (E=V-1 edges) are taken
    ii front = pq.top(); pq.pop();
    u = -front.second, w = -front.first;  // negate the id and weight again
    if (!taken[u])                    // we have not connected this vertex yet
      mst_cost += w, process(u); // take u, process all edges incident to u
  }                                            // each edge is in pq only once!
  printf("MST cost = %d (Prim's)\n", mst_cost);

  return 0;
}
```

04-BFS

```
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef pair<int, int> ii;      // In this chapter, we will frequently use these
typedef vector<ii> vii;      // three data type shortcuts. They may look cryptic
typedef vector<int> vi;   // but shortcuts are useful in competitive programming

int V, E, a, b, s;
vector<vii> AdjList;
vi p;                                      // addition: the predecessor/parent vector

void printPath(int u) {    // simple function to extract information from `vi p'
  if (u == s) { printf("%d", u); return; }
  printPath(p[u]);   // recursive call: to make the output format: s -> ... -> t
  printf(" %d", u); }

int main() {
  /*
  // Graph in Figure 4.3, format: list of unweighted edges
  // This example shows another form of reading graph input
  13 16
  0 1    1 2    2  3    0  4    1  5    2  6     3  7    5  6
  4 8    8 9    5 10    6 11    7 12    9 10    10 11   11 12
  */

  freopen("in_04.txt", "r", stdin);

  scanf("%d %d", &V, &E);

  AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to
AdjList
  for (int i = 0; i < E; i++) {
    scanf("%d %d", &a, &b);
    AdjList[a].push_back(ii(b, 0));
    AdjList[b].push_back(ii(a, 0));
  }

  // as an example, we start from this source, see Figure 4.3
  s = 5;

  // BFS routine
  // inside int main() -- we do not use recursion, thus we do not need to create
separate function!
  vi dist(V, 1000000000); dist[s] = 0;      // distance to source is 0 (default)
  queue<int> q; q.push(s);                                 // start from source
  p.assign(V, -1); // to store parent information (p must be a global variable!)
  int layer = -1;                              // for our output printing purpose
  bool isBipartite = true;       // addition of one boolean flag, initially true

  while (!q.empty()) {
    int u = q.front(); q.pop();                          // queue: layer by layer!
    if (dist[u] != layer) printf("\nLayer %d: ", dist[u]);
    layer = dist[u];
    printf("visit %d, ", u);
```

```
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                           // for each neighbors of u
      if (dist[v.first] == 1000000000) {
        dist[v.first] = dist[u] + 1;                  // v unvisited + reachable
        p[v.first] = u;          // addition: the parent of vertex v->first is u
        q.push(v.first);                              // enqueue v for next step
      }
      else if ((dist[v.first] % 2) == (dist[u] % 2))              // same parity
        isBipartite = false;
  } }

  printf("\nShortest path: ");
  printPath(7), printf("\n");
  printf("isBipartite? %d\n", isBipartite);

  return 0;
}
```

05-Dijkstra

```
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
#define INF 1000000000

int main() {
  int V, E, s, u, v, w;
  vector<vii> AdjList;

  /*
  // Graph in Figure 4.17
  5 7 2
  2 1 2
  2 3 7
  2 0 6
  1 3 3
  1 4 6
  3 4 5
  0 4 1
  */

  freopen("in_05.txt", "r", stdin);

  scanf("%d %d %d", &V, &E, &s);

  AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to
AdjList
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    AdjList[u].push_back(ii(v, w));                              // directed
graph
  }

  // Dijkstra routine
  vi dist(V, INF); dist[s] = 0;                    // INF = 1B to avoid overflow
  priority_queue< ii, vector<ii>, greater<ii> > pq; pq.push(ii(0, s));
                             // ^to sort the pairs by increasing distance from s
  while (!pq.empty()) {                                              // main loop
    ii front = pq.top(); pq.pop();     // greedy: pick shortest unvisited vertex
    int d = front.first, u = front.second;
    if (d > dist[u]) continue;   // this check is important, see the explanation
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                       // all outgoing edges from u
      if (dist[u] + v.second < dist[v.first]) {
        dist[v.first] = dist[u] + v.second;                  // relax operation
        pq.push(ii(dist[v.first], v.first));
  } } }  // note: this variant can cause duplicate items in the priority queue

  for (int i = 0; i < V; i++) // index + 1 for final answer
    printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

  return 0;
```

```
}
```

06-Bellman Ford

```
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
#define INF 1000000000

int main() {
  int V, E, s, a, b, w;
  vector<vii> AdjList;

  /*
  // Graph in Figure 4.18, has negative weight, but no negative cycle
  5 5 0
  0 1 1
  0 2 10
  1 3 2
  2 3 -10
  3 4 3

  // Graph in Figure 4.19, negative cycle exists
  3 3 0
  0 1 1000
  1 2 15
  2 1 -42
  */

  freopen("in_06.txt", "r", stdin);

  scanf("%d %d %d", &V, &E, &s);

  AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to
AdjList
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &a, &b, &w);
    AdjList[a].push_back(ii(b, w));
  }

  // Bellman Ford routine
  vi dist(V, INF); dist[s] = 0;
  for (int i = 0; i < V - 1; i++)  // relax all E edges V-1 times, overall O(VE)
    for (int u = 0; u < V; u++)                        // these two loops = O(E)
      for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];        // we can record SP spanning here if needed
        dist[v.first] = min(dist[v.first], dist[u] + v.second);         // relax
      }

  bool hasNegativeCycle = false;
  for (int u = 0; u < V; u++)                          // one more pass to check
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (dist[v.first] > dist[u] + v.second)                 // should be false
```

```
      hasNegativeCycle = true;     // but if true, then negative cycle exists!
    }
  printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");

  if (!hasNegativeCycle)
    for (int i = 0; i < V; i++)
      printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

  return 0;
}
```

07-Floyd Warshall

```
#include <algorithm>
#include <cstdio>
using namespace std;

#define INF 1000000000

int main() {
  int V, E, u, v, w, AdjMatrix[200][200];

  /*
  // Graph in Figure 4.30
  5 9
  0 1 2
  0 2 1
  0 4 3
  1 3 4
  2 1 1
  2 4 1
  3 0 1
  3 2 3
  3 4 5
  */

  freopen("in_07.txt", "r", stdin);

  scanf("%d %d", &V, &E);
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++)
      AdjMatrix[i][j] = INF;
    AdjMatrix[i][i] = 0;
  }

  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    AdjMatrix[u][v] = w; // directed graph
  }

  for (int k = 0; k < V; k++) // common error: remember that loop order is k->i->j
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        AdjMatrix[i][j] = min(AdjMatrix[i][j], AdjMatrix[i][k] + AdjMatrix[k][j]);

  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      printf("APSP(%d, %d) = %d\n", i, j, AdjMatrix[i][j]);
```

```
  return 0;
}
```

08-Edmonds Karp

```
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef vector<int> vi;

#define MAX_V 40 // enough for sample graph in Figure 4.24/4.25/4.26/UVa 259
#define INF 1000000000

int res[MAX_V][MAX_V], mf, f, s, t;                          // global variables
vi p;

void augment(int v, int minEdge) {     // traverse BFS spanning tree from s to t
  if (v == s) { f = minEdge; return; }  // record minEdge in a global variable f
  else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v])); // recursive
                         res[p[v]][v] -= f; res[v][p[v]] += f; }       // update
}

int main() {
  int V, k, vertex, weight;

  /*
  // Graph in Figure 4.24
  4 0 1
  2 2 70 3 30
  2 2 25 3 70
  3 0 70 3 5 1 25
  3 0 30 2 5 1 70

  // Graph in Figure 4.25
  4 0 3
  2 1 100 3 100
  2 2 1 3 100
  1 3 100
  0

  // Graph in Figure 4.26.A
  5 1 0
  0
  2 2 100 3 50
  3 3 50 4 50 0 50
  1 4 100
  1 0 125

  // Graph in Figure 4.26.B
  5 1 0
  0
  2 2 100 3 50
  3 3 50 4 50 0 50
  1 4 100
  1 0 75
```

```
  // Graph in Figure 4.26.C
  5 1 0
  0
  2 2 100 3 50
  2 4 5 0 5
  1 4 100
  1 0 125
  */

  freopen("in_08.txt", "r", stdin);

  scanf("%d %d %d", &V, &s, &t);

  memset(res, 0, sizeof res);
  for (int i = 0; i < V; i++) {
    scanf("%d", &k);
    for (int j = 0; j < k; j++) {
      scanf("%d %d", &vertex, &weight);
      res[i][vertex] = weight;
    }
  }

  mf = 0;                                                   // mf stands for max_flow
  while (1) {              // O(VE^2) (actually O(V^3E) Edmonds Karp's algorithm
    f = 0;
    // run BFS, compare with the original BFS shown in Section 4.2.2
    vi dist(MAX_V, INF); dist[s] = 0; queue<int> q; q.push(s);
    p.assign(MAX_V, -1);             // record the BFS spanning tree, from s to t!
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;      // immediately stop BFS if we already reach sink t
      for (int v = 0; v < MAX_V; v++)                        // note: this part is slow
        if (res[u][v] > 0 && dist[v] == INF)
          dist[v] = dist[u] + 1, q.push(v), p[v] = u;
    }
    augment(t, INF);     // find the min edge weight `f' along this path, if any
    if (f == 0) break;      // we cannot send any more flow (`f' = 0), terminate
    mf += f;                 // we can still send a flow, increase the max flow!
  }

  printf("%d\n", mf);                                  // this is the max flow value

  return 0;
}



/*

#include <algorithm>
#include <bitset>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef vector<int> vi;

#define MAX_V 40 // enough for sample graph in Figure 4.24/4.25/4.26/UVa 259
#define INF 1000000000
```

```
int res[MAX_V][MAX_V], mf, f, s, t;                          // global variables
vi p;
vector<vi> AdjList;

void augment(int v, int minEdge) {     // traverse BFS spanning tree from s to t
  if (v == s) { f = minEdge; return; }  // record minEdge in a global variable f
  else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v])); // recursive
                         res[p[v]][v] -= f; res[v][p[v]] += f; }       // update
}

int main() {
  int V, k, vertex, weight;

  scanf("%d %d %d", &V, &s, &t);

  memset(res, 0, sizeof res);
  AdjList.assign(V, vi());
  for (int i = 0; i < V; i++) {
    scanf("%d", &k);
    for (int j = 0; j < k; j++) {
      scanf("%d %d", &vertex, &weight);
      res[i][vertex] = weight;
      AdjList[i].push_back(vertex);
    }
  }

  mf = 0;
  while (1) {                     // now a true O(VE^2) Edmonds Karp's algorithm
    f = 0;
    bitset<MAX_V> vis; vis[s] = true;            // we change vi dist to bitset!
    queue<int> q; q.push(s);
    p.assign(MAX_V, -1);
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;
      for (int j = 0; j < (int)AdjList[u].size(); j++) {  // we use AdjList
here!
        int v = AdjList[u][j];
        if (res[u][v] > 0 && !vis[v])
          vis[v] = true, q.push(v), p[v] = u;
      }
    }
    augment(t, INF);
    if (f == 0) break;
    mf += f;
  }

  printf("%d\n", mf);                              // this is the max flow value

  return 0;
}
```

09-mcbm

```
*/
#include <cstdio>
#include <iostream>
#include <vector>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;

vector<vi> AdjList;
vi match, vis;                                          // global variables

int Aug(int l) {                 // return 1 if an augmenting path is found
  if (vis[l]) return 0;                               // return 0 otherwise
  vis[l] = 1;
  for (int j = 0; j < (int)AdjList[l].size(); j++) {
    int r = AdjList[l][j];
    if (match[r] == -1 || Aug(match[r])) {
      match[r] = l; return 1;                           // found 1 matching
  } }
  return 0;                                                  // no matching
}

bool isprime(int v) {
  int primes[10] = {2,3,5,7,11,13,17,19,23,29};
  for (int i = 0; i < 10; i++)
    if (primes[i] == v)
      return true;
  return false;
}

int main() {
// inside int main()
  // build bipartite graph with directed edge from left to right set

/*
  // Graph in Figure 4.40 can be built on the fly
  // we know there are 6 vertices in this bipartite graph, left side are
numbered 0,1,2, right side 3,4,5
  int V = 6, Vleft = 3, set1[3] = {1,7,11}, set2[3] = {4,10,12};

  // Graph in Figure 4.41 can be built on the fly
  // we know there are 5 vertices in this bipartite graph, left side are
numbered 0,1, right side 3,4,5
  //int V = 5, Vleft = 2, set1[2] = {1,7}, set2[3] = {4,10,12};

  // build the bipartite graph, only directed edge from left to right is needed
  AdjList.assign(V, vi());
  for (int i = 0; i < Vleft; i++)
    for (int j = 0; j < 3; j++)
      if (isprime(set1[i] + set2[j]))
        AdjList[i].push_back(3 + j);
*/

  // For bipartite graph in Figure 4.44, V = 5, Vleft = 3 (vertex 0 unused)
  // AdjList[0] = {} // dummy vertex, but you can choose to use this vertex
  // AdjList[1] = {3, 4}
```

```
  // AdjList[2] = {3}
  // AdjList[3] = {}   // we use directed edges from left to right set only
  // AdjList[4] = {}

  int V = 5, Vleft = 3;                                 // we ignore vertex 0
  AdjList.assign(V, vi());
  AdjList[1].push_back(3); AdjList[1].push_back(4);
  AdjList[2].push_back(3);

  int MCBM = 0;
  match.assign(V, -1);    // V is the number of vertices in bipartite graph
  for (int l = 0; l < Vleft; l++) {          // Vleft = size of the left set
    vis.assign(Vleft, 0);                     // reset before each recursion
    MCBM += Aug(l);
  }
  printf("Found %d matchings\n", MCBM);  // the answer is 2 for Figure 4.42

  return 0;
}
```

-------- Chapter 7: Geometria ----------
01- Points lines

```
#include <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>
using namespace std;

#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0) // important constant; alternative #define PI (2.0 *
acos(0.0))

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

// struct point_i { int x, y; };    // basic raw form, minimalist mode
struct point_i { int x, y;     // whenever possible, work with point_i
  point_i() { x = y = 0; }                     // default constructor
  point_i(int _x, int _y) : x(_x), y(_y) {} };        // user-defined

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                     // default constructor
  point(double _x, double _y) : x(_x), y(_y) {}       // user-defined
  bool operator < (point other) const { // override less than operator
    if (fabs(x - other.x) > EPS)                 // useful for sorting
      return x < other.x;          // first criteria , by x-coordinate
    return y < other.y; }          // second criteria, by y-coordinate
  // use EPS (1e-9) when testing equality of two floating points
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

double dist(point p1, point p2) {                // Euclidean distance
                      // hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
  return hypot(p1.x - p2.x, p1.y - p2.y); }           // return double

// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
  double rad = DEG_to_RAD(theta);    // multiply theta with PI / 180.0
  return point(p.x * cos(rad) - p.y * sin(rad),
               p.x * sin(rad) + p.y * cos(rad)); }

struct line { double a, b, c; };          // a way to represent a line

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
  if (fabs(p1.x - p2.x) < EPS) {              // vertical line is fine
    l.a = 1.0;    l.b = 0.0;    l.c = -p1.x;         // default values
  } else {
    l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
    l.b = 1.0;               // IMPORTANT: we fix the value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }

// not needed since we will use the more robust form: ax + by + c = 0 (see
above)
struct line2 { double m, c; };      // another way to represent a line
```

```
int pointsToLine2(point p1, point p2, line2 &l) {
 if (abs(p1.x - p2.x) < EPS) {          // special case: vertical line
   l.m = INF;                       // l contains m = INF and c = x_value
   l.c = p1.x;                     // to denote vertical line x = x_value
   return 0;   // we need this return variable to differentiate result
 }
 else {
   l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);
   l.c = p1.y - l.m * p1.x;
   return 1;     // l contains m and c of the line equation y = mx + c
} }

bool areParallel(line l1, line l2) {       // check coefficients a & b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

bool areSame(line l1, line l2) {           // also check coefficient c
  return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }

// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
  if (areParallel(l1, l2)) return false;            // no intersection
  // solve system of 2 linear algebraic equations with 2 unknowns
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  // special case: test for vertical line to avoid division by zero
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                  p.y = -(l2.a * p.x + l2.c);
  return true; }

struct vec { double x, y;  // name: `vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }               // shorter.same.longer

point translate(point p, vec v) {        // translate p according to v
  return point(p.x + v.x , p.y + v.y); }

// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line &l) {
  l.a = -m;                                                // always -m
  l.b = 1;                                                  // always 1
  l.c = -((l.a * p.x) + (l.b * p.y)); }                 // compute this

void closestPoint(line l, point p, point &ans) {
  line perpendicular;         // perpendicular to l and pass through p
  if (fabs(l.b) < EPS) {              // special case 1: vertical line
    ans.x = -(l.c);   ans.y = p.y;      return; }

  if (fabs(l.a) < EPS) {            // special case 2: horizontal line
    ans.x = p.x;      ans.y = -(l.c);   return; }

  pointSlopeToLine(p, 1 / l.a, perpendicular);          // normal line
  // intersect line l with this perpendicular line
  // the intersection point is the closest point
  areIntersect(l, perpendicular, ans); }

// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &ans) {
```

```
  point b;
  closestPoint(l, p, b);                        // similar to distToLine
  vec v = toVec(p, b);                                // create a vector
  ans = translate(translate(p, v), v); }            // translate p twice

double dot(vec a, vec b) { return (a.x * b.x + a.y * b.y); }

double norm_sq(vec v) { return v.x * v.x + v.y * v.y; }

// returns the distance from p to the line defined by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter (byref)
double distToLine(point p, point a, point b, point &c) {
  // formula: c = a + u * ab
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));                    // translate a to c
  return dist(p, c); }                  // Euclidean distance between p and c

// returns the distance from p to the line segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter (byref)
double distToLineSegment(point p, point a, point b, point &c) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  if (u < 0.0) { c = point(a.x, a.y);                         // closer to a
    return dist(p, a); }                // Euclidean distance between p and a
  if (u > 1.0) { c = point(b.x, b.y);                         // closer to b
    return dist(p, b); }                // Euclidean distance between p and b
  return distToLine(p, a, b, c); }               // run distToLine as above

double angle(point a, point o, point b) {  // returns angle aob in rad
  vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }

double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }

//// another variant
//int area2(point p, point q, point r) { // returns 'twice' the area of this
triangle A-B-c
//  return p.x * q.y - p.y * q.x +
//         q.x * r.y - q.y * r.x +
//         r.x * p.y - r.y * p.x;
//}

// note: to accept collinear points, we have to change the `> 0'
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

int main() {
  point P1, P2, P3(0, 1); // note that both P1 and P2 are (0.00, 0.00)
  printf("%d\n", P1 == P2);                                      // true
  printf("%d\n", P1 == P3);                                     // false

  vector<point> P;
  P.push_back(point(2, 2));
```

```
  P.push_back(point(4, 3));
  P.push_back(point(2, 4));
  P.push_back(point(6, 6));
  P.push_back(point(2, 6));
  P.push_back(point(6, 5));

  // sorting points demo
  sort(P.begin(), P.end());
  for (int i = 0; i < (int)P.size(); i++)
    printf("(%.2lf, %.2lf)\n", P[i].x, P[i].y);

  // rearrange the points as shown in the diagram below
  P.clear();
  P.push_back(point(2, 2));
  P.push_back(point(4, 3));
  P.push_back(point(2, 4));
  P.push_back(point(6, 6));
  P.push_back(point(2, 6));
  P.push_back(point(6, 5));
  P.push_back(point(8, 6));

  /*
  // the positions of these 7 points (0-based indexing)
  6   P4      P3  P6
  5           P5
  4   P2
  3       P1
  2   P0
  1
  0 1 2 3 4 5 6 7 8
  */

  double d = dist(P[0], P[5]);
  printf("Euclidean distance between P[0] and P[5] = %.2lf\n", d); // should be
5.000

  // line equations
  line l1, l2, l3, l4;
  pointsToLine(P[0], P[1], l1);
  printf("%.2lf * x + %.2lf * y + %.2lf = 0.00\n", l1.a, l1.b, l1.c); // should
be -0.50 * x + 1.00 * y - 1.00 = 0.00

  pointsToLine(P[0], P[2], l2); // a vertical line, not a problem in "ax + by +
c = 0" representation
  printf("%.2lf * x + %.2lf * y + %.2lf = 0.00\n", l2.a, l2.b, l2.c); // should
be 1.00 * x + 0.00 * y - 2.00 = 0.00

  // parallel, same, and line intersection tests
  pointsToLine(P[2], P[3], l3);
  printf("l1 & l2 are parallel? %d\n", areParallel(l1, l2)); // no
  printf("l1 & l3 are parallel? %d\n", areParallel(l1, l3)); // yes, l1 (P[0]-
P[1]) and l3 (P[2]-P[3]) are parallel

  pointsToLine(P[2], P[4], l4);
  printf("l1 & l2 are the same? %d\n", areSame(l1, l2)); // no
  printf("l2 & l4 are the same? %d\n", areSame(l2, l4)); // yes, l2 (P[0]-P[2])
and l4 (P[2]-P[4]) are the same line (note, they are two different line
segments, but same line)

  point p12;
```

```
  bool res = areIntersect(l1, l2, p12); // yes, l1 (P[0]-P[1]) and l2 (P[0]-
P[2]) are intersect at (2.0, 2.0)
  printf("l1 & l2 are intersect? %d, at (%.2lf, %.2lf)\n", res, p12.x, p12.y);

  // other distances
  point ans;
  d = distToLine(P[0], P[2], P[3], ans);
  printf("Closest point from P[0] to line         (P[2]-P[3]): (%.2lf, %.2lf),
dist = %.2lf\n", ans.x, ans.y, d);
  closestPoint(l3, P[0], ans);
  printf("Closest point from P[0] to line V2      (P[2]-P[3]): (%.2lf, %.2lf),
dist = %.2lf\n", ans.x, ans.y, dist(P[0], ans));

  d = distToLineSegment(P[0], P[2], P[3], ans);
  printf("Closest point from P[0] to line SEGMENT (P[2]-P[3]): (%.2lf, %.2lf),
dist = %.2lf\n", ans.x, ans.y, d); // closer to A (or P[2]) = (2.00, 4.00)
  d = distToLineSegment(P[1], P[2], P[3], ans);
  printf("Closest point from P[1] to line SEGMENT (P[2]-P[3]): (%.2lf, %.2lf),
dist = %.2lf\n", ans.x, ans.y, d); // closer to midway between AB = (3.20, 4.60)
  d = distToLineSegment(P[6], P[2], P[3], ans);
  printf("Closest point from P[6] to line SEGMENT (P[2]-P[3]): (%.2lf, %.2lf),
dist = %.2lf\n", ans.x, ans.y, d); // closer to B (or P[3]) = (6.00, 6.00)

  reflectionPoint(l4, P[1], ans);
  printf("Reflection point from P[1] to line      (P[2]-P[4]): (%.2lf,
%.2lf)\n", ans.x, ans.y); // should be (0.00, 3.00)

  printf("Angle P[0]-P[4]-P[3] = %.2lf\n", RAD_to_DEG(angle(P[0], P[4], P[3])));
// 90 degrees
  printf("Angle P[0]-P[2]-P[1] = %.2lf\n", RAD_to_DEG(angle(P[0], P[2], P[1])));
// 63.43 degrees
  printf("Angle P[4]-P[3]-P[6] = %.2lf\n", RAD_to_DEG(angle(P[4], P[3], P[6])));
// 180 degrees

  printf("P[0], P[2], P[3] form A left turn? %d\n", ccw(P[0], P[2], P[3])); //
no
  printf("P[0], P[3], P[2] form A left turn? %d\n", ccw(P[0], P[3], P[2])); //
yes

  printf("P[0], P[2], P[3] are collinear? %d\n", collinear(P[0], P[2], P[3]));
// no
  printf("P[0], P[2], P[4] are collinear? %d\n", collinear(P[0], P[2], P[4]));
// yes

  point p(3, 7), q(11, 13), r(35, 30); // collinear if r(35, 31)
  printf("r is on the %s of line p-r\n", ccw(p, q, r) ? "left" : "right"); //
right

  /*
  // the positions of these 6 points
     E<--  4
           3       B D<--
           2   A C
           1
  -4-3-2-1 0 1 2 3 4 5 6
          -1
          -2
   F<--   -3
  */

  // translation
```

```
  point A(2.0, 2.0);
  point B(4.0, 3.0);
  vec v = toVec(A, B); // imagine there is an arrow from A to B (see the diagram
above)
  point C(3.0, 2.0);
  point D = translate(C, v); // D will be located in coordinate (3.0 + 2.0, 2.0
+ 1.0) = (5.0, 3.0)
  printf("D = (%.2lf, %.2lf)\n", D.x, D.y);
  point E = translate(C, scale(v, 0.5)); // E will be located in coordinate (3.0
+ 1/2 * 2.0, 2.0 + 1/2 * 1.0) = (4.0, 2.5)
  printf("E = (%.2lf, %.2lf)\n", E.x, E.y);

  // rotation
  printf("B = (%.2lf, %.2lf)\n", B.x, B.y); // B = (4.0, 3.0)
  point F = rotate(B, 90); // rotate B by 90 degrees COUNTER clockwise, F = (-
3.0, 4.0)
  printf("F = (%.2lf, %.2lf)\n", F.x, F.y);
  point G = rotate(B, 180); // rotate B by 180 degrees COUNTER clockwise, G = (-
4.0, -3.0)
  printf("G = (%.2lf, %.2lf)\n", G.x, G.y);

  return 0;
}
```

---

02 - Circles

```
#include <cstdio>
#include <cmath>
using namespace std;

#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0)

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

struct point_i { int x, y;     // whenever possible, work with point_i
  point_i() { x = y = 0; }                     // default constructor
  point_i(int _x, int _y) : x(_x), y(_y) {} };         // constructor

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                     // default constructor
  point(double _x, double _y) : x(_x), y(_y) {} };     // constructor

int insideCircle(point_i p, point_i c, int r) { // all integer version
  int dx = p.x - c.x, dy = p.y - c.y;
  int Euc = dx * dx + dy * dy, rSq = r * r;             // all integer
  return Euc < rSq ? 0 : Euc == rSq ? 1 : 2; } //inside/border/outside

bool circle2PtsRad(point p1, point p2, double r, point &c) {
  double d2 = (p1.x - p2.x) * (p1.x - p2.x) +
              (p1.y - p2.y) * (p1.y - p2.y);
  double det = r * r / d2 - 0.25;
  if (det < 0.0) return false;
  double h = sqrt(det);
  c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
  c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
  return true; }         // to get the other center, reverse p1 and p2
```

```
int main() {
  // circle equation, inside, border, outside
  point_i pt(2, 2);
  int r = 7;
  point_i inside(8, 2);
  printf("%d\n", insideCircle(inside, pt, r));             // 0-inside
  point_i border(9, 2);
  printf("%d\n", insideCircle(border, pt, r));          // 1-at border
  point_i outside(10, 2);
  printf("%d\n", insideCircle(outside, pt, r));           // 2-outside

  double d = 2 * r;
  printf("Diameter = %.2lf\n", d);
  double c = PI * d;
  printf("Circumference (Perimeter) = %.2lf\n", c);
  double A = PI * r * r;
  printf("Area of circle = %.2lf\n", A);

  printf("Length of arc   (central angle = 60 degrees) = %.2lf\n", 60.0 / 360.0
* c);
  printf("Length of chord (central angle = 60 degrees) = %.2lf\n", sqrt((2 * r *
r) * (1 - cos(DEG_to_RAD(60.0)))));
  printf("Area of sector  (central angle = 60 degrees) = %.2lf\n", 60.0 / 360.0
* A);

  point p1;
  point p2(0.0, -1.0);
  point ans;
  circle2PtsRad(p1, p2, 2.0, ans);
  printf("One of the center is (%.2lf, %.2lf)\n", ans.x, ans.y);
  circle2PtsRad(p2, p1, 2.0, ans);     // we simply reverse p1 with p2
  printf("The other center  is (%.2lf, %.2lf)\n", ans.x, ans.y);

  return 0;
}
```

---

03 - Triangles

```
#include <cstdio>
#include <cmath>
using namespace std;

#define EPS 1e-9
#define PI acos(-1.0)

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

struct point_i { int x, y;     // whenever possible, work with point_i
  point_i() { x = y = 0; }                        // default constructor
  point_i(int _x, int _y) : x(_x), y(_y) {} };           // constructor

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                        // default constructor
  point(double _x, double _y) : x(_x), y(_y) {} };       // constructor

double dist(point p1, point p2) {
```

```
  return hypot(p1.x - p2.x, p1.y - p2.y); }

double perimeter(double ab, double bc, double ca) {
  return ab + bc + ca; }

double perimeter(point a, point b, point c) {
  return dist(a, b) + dist(b, c) + dist(c, a); }

double area(double ab, double bc, double ca) {
  // Heron's formula, split sqrt(a * b) into sqrt(a) * sqrt(b); in
implementation
  double s = 0.5 * perimeter(ab, bc, ca);
  return sqrt(s) * sqrt(s - ab) * sqrt(s - bc) * sqrt(s - ca); }

double area(point a, point b, point c) {
  return area(dist(a, b), dist(b, c), dist(c, a)); }

//====================================================================
// from ch7_01_points_lines
struct line { double a, b, c; }; // a way to represent a line

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
  if (fabs(p1.x - p2.x) < EPS) {              // vertical line is fine
    l.a = 1.0;   l.b = 0.0;   l.c = -p1.x;           // default values
  } else {
    l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
    l.b = 1.0;              // IMPORTANT: we fix the value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }

bool areParallel(line l1, line l2) {        // check coefficient a + b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
  if (areParallel(l1, l2)) return false;            // no intersection
  // solve system of 2 linear algebraic equations with 2 unknowns
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  // special case: test for vertical line to avoid division by zero
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                  p.y = -(l2.a * p.x + l2.c);
  return true; }

struct vec { double x, y;  // name: `vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }               // shorter.same.longer

point translate(point p, vec v) {        // translate p according to v
  return point(p.x + v.x , p.y + v.y); }
//====================================================================

double rInCircle(double ab, double bc, double ca) {
  return area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca)); }

double rInCircle(point a, point b, point c) {
```

```
  return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written
// returns 1 if there is an inCircle center, returns 0 otherwise
// if this function returns 1, ctr will be the inCircle center
// and r is the same as rInCircle
int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
  r = rInCircle(p1, p2, p3);
  if (fabs(r) < EPS) return 0;                   // no inCircle center

  line l1, l2;                    // compute these two angle bisectors
  double ratio = dist(p1, p2) / dist(p1, p3);
  point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
  pointsToLine(p1, p, l1);

  ratio = dist(p2, p1) / dist(p2, p3);
  p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
  pointsToLine(p2, p, l2);

  areIntersect(l1, l2, ctr);           // get their intersection point
  return 1; }

double rCircumCircle(double ab, double bc, double ca) {
  return ab * bc * ca / (4.0 * area(ab, bc, ca)); }

double rCircumCircle(point a, point b, point c) {
  return rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written
// returns 1 if there is a circumCenter center, returns 0 otherwise
// if this function returns 1, ctr will be the circumCircle center
// and r is the same as rCircumCircle
int circumCircle(point p1, point p2, point p3, point &ctr, double &r){
  double a = p2.x - p1.x, b = p2.y - p1.y;
  double c = p3.x - p1.x, d = p3.y - p1.y;
  double e = a * (p1.x + p2.x) + b * (p1.y + p2.y);
  double f = c * (p1.x + p3.x) + d * (p1.y + p3.y);
  double g = 2.0 * (a * (p3.y - p2.y) - b * (p3.x - p2.x));
  if (fabs(g) < EPS) return 0;

  ctr.x = (d*e - b*f) / g;
  ctr.y = (a*f - c*e) / g;
  r = dist(p1, ctr);  // r = distance from center to 1 of the 3 points
  return 1; }

// returns true if point d is inside the circumCircle defined by a,b,c
int inCircumCircle(point a, point b, point c, point d) {
  return (a.x - d.x) * (b.y - d.y) * ((c.x - d.x) * (c.x - d.x) + (c.y - d.y) *
(c.y - d.y)) +
         (a.y - d.y) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y - d.y)) *
(c.x - d.x) +
         ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.x - d.x) *
(c.y - d.y) -
         ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.y - d.y) *
(c.x - d.x) -
         (a.y - d.y) * (b.x - d.x) * ((c.x - d.x) * (c.x - d.x) + (c.y - d.y) *
(c.y - d.y)) -
         (a.x - d.x) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y - d.y)) *
(c.y - d.y) > 0 ? 1 : 0;
}
```

```
bool canFormTriangle(double a, double b, double c) {
  return (a + b > c) && (a + c > b) && (b + c > a); }

int main() {
  double base = 4.0, h = 3.0;
  double A = 0.5 * base * h;
  printf("Area = %.2lf\n", A);

  point a;                                          // a right triangle
  point b(4.0, 0.0);
  point c(4.0, 3.0);

  double p = perimeter(a, b, c);
  double s = 0.5 * p;
  A = area(a, b, c);
  printf("Area = %.2lf\n", A);              // must be the same as above

  double r = rInCircle(a, b, c);
  printf("R1 (radius of incircle) = %.2lf\n", r);              // 1.00
  point ctr;
  int res = inCircle(a, b, c, ctr, r);
  printf("R1 (radius of incircle) = %.2lf\n", r);        // same, 1.00
  printf("Center = (%.2lf, %.2lf)\n", ctr.x, ctr.y);   // (3.00, 1.00)

  printf("R2 (radius of circumcircle) = %.2lf\n", rCircumCircle(a, b, c)); //
2.50
  res = circumCircle(a, b, c, ctr, r);
  printf("R2 (radius of circumcircle) = %.2lf\n", r);   // same, 2.50
  printf("Center = (%.2lf, %.2lf)\n", ctr.x, ctr.y);   // (2.00, 1.50)

  point d(2.0, 1.0);                // inside triangle and circumCircle
  printf("d inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b, c, d));
  point e(2.0, 3.9);   // outside the triangle but inside circumCircle
  printf("e inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b, c, e));
  point f(2.0, -1.1);                               // slightly outside
  printf("f inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b, c, f));

  // Law of Cosines
  double ab = dist(a, b);
  double bc = dist(b, c);
  double ca = dist(c, a);
  double alpha = RAD_to_DEG(acos((ca * ca + ab * ab - bc * bc) / (2.0 * ca *
ab)));
  printf("alpha = %.2lf\n", alpha);
  double beta  = RAD_to_DEG(acos((ab * ab + bc * bc - ca * ca) / (2.0 * ab *
bc)));
  printf("beta  = %.2lf\n", beta);
  double gamma = RAD_to_DEG(acos((bc * bc + ca * ca - ab * ab) / (2.0 * bc *
ca)));
  printf("gamma = %.2lf\n", gamma);

  // Law of Sines
  printf("%.2lf == %.2lf == %.2lf\n", bc / sin(DEG_to_RAD(alpha)), ca /
sin(DEG_to_RAD(beta)), ab / sin(DEG_to_RAD(gamma)));

  // Phytagorean Theorem
  printf("%.2lf^2 == %.2lf^2 + %.2lf^2\n", ca, ab, bc);

  // Triangle Inequality
  printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 5, canFormTriangle(3,
4, 5)); // yes
```

```
  printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 7, canFormTriangle(3,
4, 7)); // no, actually straight line
  printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 8, canFormTriangle(3,
4, 8)); // no

  return 0;
}
```

---

04 - Polygon

```
#include <algorithm>
#include <cstdio>
#include <cmath>
#include <stack>
#include <vector>
using namespace std;

#define EPS 1e-9
#define PI acos(-1.0)

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                      // default constructor
  point(double _x, double _y) : x(_x), y(_y) {}        // user-defined
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

struct vec { double x, y;  // name: `vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

double dist(point p1, point p2) {                // Euclidean distance
  return hypot(p1.x - p2.x, p1.y - p2.y); }           // return double

// returns the perimeter, which is the sum of Euclidian distances
// of consecutive line segments (polygon edges)
double perimeter(const vector<point> &P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++)  // remember that P[0] = P[n-1]
    result += dist(P[i], P[i+1]);
  return result; }

// returns the area, which is half the determinant
double area(const vector<point> &P) {
  double result = 0.0, x1, y1, x2, y2;
  for (int i = 0; i < (int)P.size()-1; i++) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1 * y2 - x2 * y1);
  }
  return fabs(result) / 2.0; }

double dot(vec a, vec b) { return (a.x * b.x + a.y * b.y); }
```

```
double norm_sq(vec v) { return v.x * v.x + v.y * v.y; }

double angle(point a, point o, point b) {  // returns angle aob in rad
  vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }

double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }

// note: to accept collinear points, we have to change the `> 0'
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

// returns true if we always make the same turn while examining
// all the edges of the polygon one by one
bool isConvex(const vector<point> &P) {
  int sz = (int)P.size();
  if (sz <= 3) return false;   // a point/sz=2 or a line/sz=3 is not convex
  bool isLeft = ccw(P[0], P[1], P[2]);               // remember one result
  for (int i = 1; i < sz-1; i++)            // then compare with the others
    if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
      return false;            // different sign -> this polygon is concave
  return true; }                                   // this polygon is convex

// returns true if point p is in either convex/concave polygon P
bool inPolygon(point pt, const vector<point> &P) {
  if ((int)P.size() == 0) return false;
  double sum = 0;    // assume the first vertex is equal to the last vertex
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
         sum += angle(P[i], pt, P[i+1]);                   // left turn/ccw
    else sum -= angle(P[i], pt, P[i+1]); }                 // right turn/cw
  return fabs(fabs(sum) - 2*PI) < EPS; }

// line segment p-q intersect with line A-B.
point lineIntersectSeg(point p, point q, point A, point B) {
  double a = B.y - A.y;
  double b = A.x - B.x;
  double c = B.x * A.y - A.x * B.y;
  double u = fabs(a * p.x + b * p.y + c);
  double v = fabs(a * q.x + b * q.y + c);
  return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u) / (u+v)); }

// cuts polygon Q along the line formed by point a -> point b
// (note: the last point must be the same as the first point)
vector<point> cutPolygon(point a, point b, const vector<point> &Q) {
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
    if (left1 > -EPS) P.push_back(Q[i]);       // Q[i] is on the left of ab
    if (left1 * left2 < -EPS)        // edge (Q[i], Q[i+1]) crosses line ab
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
  }
  if (!P.empty() && !(P.back() == P.front()))
    P.push_back(P.front());        // make P's first point = P's last point
  return P; }
```

```cpp
point pivot;
bool angleCmp(point a, point b) {                 // angle-sorting function
  if (collinear(pivot, a, b))                               // special case
    return dist(pivot, a) < dist(pivot, b);    // check which one is closer
  double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; }   // compare two angles

vector<point> CH(vector<point> P) {   // the content of P may be reshuffled
  int i, j, n = (int)P.size();
  if (n <= 3) {
    if (!(P[0] == P[n-1])) P.push_back(P[0]); // safeguard from corner case
    return P;                           // special case, the CH is P itself
  }

  // first, find P0 = point with lowest Y and if tie: rightmost X
  int P0 = 0;
  for (i = 1; i < n; i++)
    if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))
      P0 = i;

  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;    // swap P[P0] with P[0]

  // second, sort points by angle w.r.t. pivot P0
  pivot = P[0];                    // use this global variable as reference
  sort(++P.begin(), P.end(), angleCmp);              // we do not sort P[0]

  // third, the ccw tests
  vector<point> S;
  S.push_back(P[n-1]); S.push_back(P[0]); S.push_back(P[1]);   // initial S
  i = 2;                                         // then, we check the rest
  while (i < n) {           // note: N must be >= 3 for this method to work
    j = (int)S.size()-1;
    if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]);  // left turn, accept
    else S.pop_back(); }   // or pop the top of S until we have a left turn
  return S; }                                          // return the result

int main() {
  // 6 points, entered in counter clockwise order, 0-based indexing
  vector<point> P;
  P.push_back(point(1, 1));
  P.push_back(point(3, 3));
  P.push_back(point(9, 1));
  P.push_back(point(12, 4));
  P.push_back(point(9, 7));
  P.push_back(point(1, 7));
  P.push_back(P[0]); // loop back

  printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // 31.64
  printf("Area of polygon = %.2lf\n", area(P)); // 49.00
  printf("Is convex = %d\n", isConvex(P)); // false (P1 is the culprit)

  //// the positions of P6 and P7 w.r.t the polygon
  //7 P5--------------P4
  //6 |                  \
  //5 |                    \
  //4 |   P7                P3
  //3 |   P1___            /
  //2 | / P6    \ ___    /
  //1 P0              P2
```

```
  //0 1 2 3 4 5 6 7 8 9 101112

  point P6(3, 2); // outside this (concave) polygon
  printf("Point P6 is inside this polygon = %d\n", inPolygon(P6, P)); // false
  point P7(3, 4); // inside this (concave) polygon
  printf("Point P7 is inside this polygon = %d\n", inPolygon(P7, P)); // true

  // cutting the original polygon based on line P[2] -> P[4] (get the left side)
  //7 P5--------------P4
  //6 |               |  \
  //5 |               |    \
  //4 |               |     P3
  //3 |   P1___       |    /
  //2 | /       \ ___ |  /
  //1 P0              P2
  //0 1 2 3 4 5 6 7 8 9 101112
  // new polygon (notice the index are different now):
  //7 P4--------------P3
  //6 |               |
  //5 |               |
  //4 |               |
  //3 |   P1___       |
  //2 | /       \ ___ |
  //1 P0              P2
  //0 1 2 3 4 5 6 7 8 9

  P = cutPolygon(P[2], P[4], P);
  printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // smaller now 29.15
  printf("Area of polygon = %.2lf\n", area(P)); // 40.00

  // running convex hull of the resulting polygon (index changes again)
  //7 P3--------------P2
  //6 |               |
  //5 |               |
  //4 |   P7          |
  //3 |               |
  //2 |               |
  //1 P0--------------P1
  //0 1 2 3 4 5 6 7 8 9

  P = CH(P); // now this is a rectangle
  printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // precisely 28.00
  printf("Area of polygon = %.2lf\n", area(P)); // precisely 48.00
  printf("Is convex = %d\n", isConvex(P)); // true
  printf("Point P6 is inside this polygon = %d\n", inPolygon(P6, P)); // true
  printf("Point P7 is inside this polygon = %d\n", inPolygon(P7, P)); // true

  return 0;
}
```