Contents

Template

模数表:

	1	2	3	4	5
1e6	1000003	1000033	1000037	1000621	1000621
1e9	1000000007	1000000009	1000000021	1000000993	1000001011
1e9	1000000007	1000000009	1000000021	1000000993	1000001011
1e14	100000000000031	100000000000067	100000000000097	100000000001623	100000000001647

单调队列

```cpp
#include <cstdio>
#include <deque>
#include <vector>

class MonotoneQueue {
  private:
    struct QueueItem {
        int data;
        size_t idx;

        explicit constexpr QueueItem(int data_, size_t idx_)
            : data{data_}, idx{idx_} {};
    };

    std::deque<QueueItem> queue;
    size_t window_length;
    bool increasing;

    size_t idx;

  public:
    explicit MonotoneQueue(size_t window_length_, bool increasing_)
        : window_length{window_length_}, increasing{increasing_}, idx
           {0} {}

    void put(int data) {
        idx++;

        while (!queue.empty() && (idx - queue.front().idx) >=
           window_length) {
            queue.pop_front();
```

```
        }

        if (increasing) {
            while (!queue.empty() && queue.back().data > data) {
                queue.pop_back();
            }
        } else {
            while (!queue.empty() && queue.back().data < data) {
                queue.pop_back();
            }
        }

        queue.emplace_back(data, idx);
    }

    int get() const { return queue.front().data; }
};

int main(void) {
    size_t n, k;
    n = k = 0;

    std::scanf("%zu%zu", &n, &k);

    std::vector<int> input_list;
    input_list.reserve(n);

    for (size_t i = 0; i < n; i++) {
        int data = 0;
        std::scanf("%d", &data);
        input_list.push_back(data);
    }

    MonotoneQueue min_queue{k, true};
    MonotoneQueue max_queue{k, false};

    for (size_t i = 0; i < n; i++) {
        min_queue.put(input_list[i]);

        if (i >= (k - 1)) {
            std::printf("%d ", min_queue.get());
        }
    }

    std::putchar('\n');

    for (size_t i = 0; i < n; i++) {
        max_queue.put(input_list[i]);

        if (i >= (k - 1)) {
            std::printf("%d ", max_queue.get());
```

```
        }
    }

    std::putchar('\n');

    return 0;
}
```

大顶堆

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

typedef long long i64;

class Heap {
  public:
    std::vector<i64> heap;
    size_t heap_size;

    explicit Heap() { heap_size = 0; }

    explicit Heap(std::vector<i64> const &num_list) {
        heap = num_list;
        heap_size = num_list.size();

        if (num_list.size() >= 2) {
            size_t idx = num_list.size() / 2 - 1;
            for (size_t i = 0; i <= num_list.size() / 2 - 1; i++) {
                max_heapify(idx);

                idx--;
            }
        }
    }

  public:
    i64 heap_maximum() { return heap[0]; }
    i64 heap_extract_max() {
        i64 max_value = heap[0];

        heap[0] = heap[heap_size - 1];

        max_heapify(0);

        heap_size--;
        return max_value;
    }

```

```
40      void heap_increase_key(size_t i, i64 new_key) {
41          heap[i] = new_key;
42
43          while (i >= 1 && heap[parent(i)] < heap[i]) {
44              std::swap(heap[parent(i)], heap[i]);
45
46              i = parent(i);
47          }
48      }
49
50      void heap_insert(i64 key) {
51          heap.push_back(0);
52
53          heap_increase_key(heap_size - 1, key);
54          heap_size++;
55      }
56
57    private:
58      inline size_t left(size_t i) { return 2 * i + 1; }
59
60      inline size_t right(size_t i) { return 2 * i + 2; }
61
62      inline size_t parent(size_t i) { return (i - 1) / 2; }
63
64      void max_heapify(size_t i) {
65          while (1) {
66              size_t left_idx = left(i);
67              size_t right_idx = right(i);
68
69              size_t largest_idx = i;
70
71              if (left_idx < heap_size && heap[left_idx] > heap[i]) {
72                  largest_idx = left_idx;
73              }
74
75              if (right_idx < heap_size && heap[right_idx] > heap[
                    largest_idx]) {
76                  largest_idx = right_idx;
77              }
78
79              if (largest_idx != i) {
80                  std::swap(heap[i], heap[largest_idx]);
81
82                  i = largest_idx;
83              } else {
84                  break;
85              }
86          }
87      }
88  };
```

排序

归并排序

```cpp
#include <algorithm>
#include <cstdio>
#include <vector>

typedef long long i64;

void merge(std::vector<i64> &arr, size_t p, size_t q, size_t r);
void do_merge_sort(std::vector<i64> &arr, size_t p, size_t r);
void merge_sort(std::vector<i64> &arr);

int main(void) {
    size_t n = 0;
    std::scanf("%zu", &n);

    std::vector<i64> num_list;

    num_list.reserve(n);

    for (size_t i = 0; i < n; i++) {
        i64 num = 0;
        std::scanf("%lld", &num);
        num_list.push_back(num);
    }

    merge_sort(num_list);

    for (i64 num : num_list) {
        std::printf("%lld ", num);
    }

    std::putchar('\n');

    return 0;
}

// merge sorted [p, q] and [q + 1, r]
void merge(std::vector<i64> &arr, size_t p, size_t q, size_t r) {
    std::vector<i64> aux;
    aux.reserve(r - p + 1);

    size_t i = p;
    size_t j = q + 1;

    while ((i <= q) && (j <= r)) {
        if (arr[i] <= arr[j]) {
            aux.push_back(arr[i]);
```

```
            i++;
        } else {
            aux.push_back(arr[j]);
            j++;
        }
    }

    while (i <= q) {
        aux.push_back(arr[i]);
        i++;
    }

    while (j <= r) {
        aux.push_back(arr[j]);
        j++;
    }

    std::copy(std::begin(aux), std::end(aux), std::begin(arr) + (long)p
        );
}

void do_merge_sort(std::vector<i64> &arr, size_t p, size_t r) {
    if (p == r) {
        return;
    }

    size_t q = p + (r - p) / 2;

    do_merge_sort(arr, p, q);
    do_merge_sort(arr, q + 1, r);

    merge(arr, p, q, r);
}

void merge_sort(std::vector<i64> &arr) {
    if (arr.empty() || arr.size() == 1) {
        return;
    }

    do_merge_sort(arr, 0, arr.size() - 1);
}
```

堆排序

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

typedef long long i64;
```

```cpp

class Heap {
  public:
    std::vector<i64> heap;
    size_t heap_size;

    explicit Heap() { heap_size = 0; }

    explicit Heap(std::vector<i64> const &num_list) {
        heap = num_list;
        heap_size = num_list.size();

        if (num_list.size() >= 2) {
            size_t idx = num_list.size() / 2 - 1;
            for (size_t i = 0; i <= num_list.size() / 2 - 1; i++) {
                max_heapify(idx);

                idx--;
            }
        }
    }

  public:
    i64 heap_maximum() { return heap[0]; }
    i64 heap_extract_max() {
        i64 max_value = heap[0];

        heap[0] = heap[heap_size - 1];

        max_heapify(0);

        heap_size--;
        return max_value;
    }

    void heap_increase_key(size_t i, i64 new_key) {
        heap[i] = new_key;

        while (i >= 1 && heap[parent(i)] < heap[i]) {
            std::swap(heap[parent(i)], heap[i]);

            i = parent(i);
        }
    }

    void heap_insert(i64 key) {
        heap.push_back(0);

        heap_increase_key(heap_size - 1, key);
        heap_size++;
    }
```

```
57
58      void max_heapify(size_t i) {
59          while (1) {
60              size_t left_idx = left(i);
61              size_t right_idx = right(i);
62
63              size_t largest_idx = i;
64
65              if (left_idx < heap_size && heap[left_idx] > heap[i]) {
66                  largest_idx = left_idx;
67              }
68
69              if (right_idx < heap_size && heap[right_idx] > heap[
                    largest_idx]) {
70                  largest_idx = right_idx;
71              }
72
73              if (largest_idx != i) {
74                  std::swap(heap[i], heap[largest_idx]);
75
76                  i = largest_idx;
77              } else {
78                  break;
79              }
80          }
81      }
82
83    private:
84      inline size_t left(size_t i) { return 2 * i + 1; }
85
86      inline size_t right(size_t i) { return 2 * i + 2; }
87
88      inline size_t parent(size_t i) { return (i - 1) / 2; }
89  };
90
91  std::vector<i64> heap_sort(std::vector<i64> const &list);
92
93  int main(void) {
94      size_t n = 0;
95      std::scanf("%zu", &n);
96
97      std::vector<i64> num_list;
98
99      num_list.reserve(n);
100
101     for (size_t i = 0; i < n; i++) {
102         i64 num = 0;
103         std::scanf("%lld", &num);
104         num_list.push_back(num);
105     }
106
```

```
    std::vector<i64> sorted = heap_sort(num_list);

    for (i64 num : sorted) {
        std::printf("%lld ", num);
    }

    std::putchar('\n');

    return 0;
}

std::vector<i64> heap_sort(std::vector<i64> const &list) {
    Heap heap{list};

    if (list.empty()) {
        return list;
    }

    for (size_t i = 0; i < list.size() - 1; i++) {
        std::swap(heap.heap[0], heap.heap[list.size() - 1 - i]);

        heap.heap_size--;

        heap.max_heapify(0);
    }

    return heap.heap;
}
```

快速排序

```
#include <cstdio>
#include <random>
#include <vector>

typedef long long i64;

i64 partition(std::vector<i64> &list, i64 p, i64 r);
void do_quick_sort(std::vector<i64> &list, i64 p, i64 r);
void quick_sort(std::vector<i64> &list);
void random_permutation(std::vector<i64> &list);

int main(void) {
    size_t n = 0;
    std::scanf("%zu", &n);

    std::vector<i64> num_list;

    num_list.reserve(n);
```

```
19
20      for (size_t i = 0; i < n; i++) {
21          i64 num = 0;
22          std::scanf("%lld", &num);
23          num_list.push_back(num);
24      }
25
26      quick_sort(num_list);
27
28      for (i64 num : num_list) {
29          std::printf("%lld ", num);
30      }
31
32      std::putchar('\n');
33
34      return 0;
35  }
36
37  // partition range [p, r]
38  i64 partition(std::vector<i64> &list, i64 p, i64 r) {
39      i64 pivot = list[(size_t)r];
40
41      i64 i = p - 1;
42
43      for (i64 j = p; j <= r - 1; j++) {
44          if (list[(size_t)j] <= pivot) {
45              i++;
46
47              std::swap(list[(size_t)i], list[(size_t)j]);
48          }
49      }
50
51      std::swap(list[(size_t)i + 1], list[(size_t)r]);
52
53      return i + 1;
54  }
55
56  void do_quick_sort(std::vector<i64> &list, i64 p, i64 r) {
57      if (p < r) {
58          i64 q = partition(list, p, r);
59
60          do_quick_sort(list, p, q - 1);
61          do_quick_sort(list, q + 1, r);
62      }
63  }
64
65  void quick_sort(std::vector<i64> &list) {
66      if (list.empty()) {
67          return;
68      }
69      random_permutation(list);
```

```
    do_quick_sort(list, 0, (i64)(list.size() - 1));
}

void random_permutation(std::vector<i64> &list) {
    auto const seed = 84841984;

    std::mt19937 urbg{seed};

    for (size_t i = 0; i < list.size(); i++) {
        std::uniform_int_distribution<size_t> distr{i, list.size() -
            1};

        std::swap(list[i], list[distr(urbg)]);
    }
}
```

哈希

```
#include <stddef.h>
#include <stdint.h>
#include <string.h>

#define SIPHASH_COMPRESSION_ROUND 2
#define SIPHASH_FINALIZATION_ROUND 4

#define ROTL(x, b) (uint64_t)(((x) << (b)) | ((x) >> (64 - (b))))

uint64_t siphash(uint64_t k_0, uint64_t k_1, void *msg, size_t msg_len)
    ;

uint64_t siphash(uint64_t k_0, uint64_t k_1, void *msg, size_t msg_len)
     {
    uint64_t v_0 = k_0 ^ 0x736f6d6570736575;
    uint64_t v_1 = k_1 ^ 0x646f72616e646f6d;
    uint64_t v_2 = k_0 ^ 0x6c7967656e657261;
    uint64_t v_3 = k_1 ^ 0x7465646279746573;

    size_t normal_count = msg_len / 8;

    uint64_t *current_word_ptr = (uint64_t *)msg;

    for (size_t i = 0; i < normal_count; i++) {
        uint64_t current_word = *current_word_ptr;

        v_3 ^= current_word;

        for (int j = 0; j < SIPHASH_COMPRESSION_ROUND; j++) {
            // SipRound
            {
```

```
                v_0 += v_1;
                v_2 += v_3;

                v_1 = ROTL(v_1, 13);
                v_3 = ROTL(v_3, 16);

                v_1 ^= v_0;
                v_3 ^= v_2;

                v_0 = ROTL(v_0, 32);

                v_2 += v_1;
                v_0 += v_3;

                v_1 = ROTL(v_1, 17);
                v_3 = ROTL(v_3, 21);

                v_1 ^= v_2;
                v_3 ^= v_0;

                v_2 = ROTL(v_2, 32);
            }
        }

        v_0 ^= current_word;

        current_word_ptr++;
    }

    uint64_t final_word = 0;

    memcpy(&final_word, current_word_ptr, msg_len - normal_count * 8);

    final_word |= (msg_len % 256) << 56;

    v_3 ^= final_word;

    for (int j = 0; j < SIPHASH_COMPRESSION_ROUND; j++) {
        // SipRound
        {
            v_0 += v_1;
            v_2 += v_3;

            v_1 = ROTL(v_1, 13);
            v_3 = ROTL(v_3, 16);

            v_1 ^= v_0;
            v_3 ^= v_2;

            v_0 = ROTL(v_0, 32);

```

```
            v_2 += v_1;
            v_0 += v_3;

            v_1 = ROTL(v_1, 17);
            v_3 = ROTL(v_3, 21);

            v_1 ^= v_2;
            v_3 ^= v_0;

            v_2 = ROTL(v_2, 32);
        }
    }

    v_0 ^= final_word;

    v_2 ^= 0xff;

    for (int j = 0; j < SIPHASH_FINALIZATION_ROUND; j++) {
        // SipRound
        {
            v_0 += v_1;
            v_2 += v_3;

            v_1 = ROTL(v_1, 13);
            v_3 = ROTL(v_3, 16);

            v_1 ^= v_0;
            v_3 ^= v_2;

            v_0 = ROTL(v_0, 32);

            v_2 += v_1;
            v_0 += v_3;

            v_1 = ROTL(v_1, 17);
            v_3 = ROTL(v_3, 21);

            v_1 ^= v_2;
            v_3 ^= v_0;

            v_2 = ROTL(v_2, 32);
        }
    }

    return v_0 ^ v_1 ^ v_2 ^ v_3;
}

int main(void) {

    uint64_t k_0 = 0x0706050403020100;
    uint64_t k_1 = 0x0f0e0d0c0b0a0908;
```

```
132
133     uint8_t buf[15] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
134                        0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e};
135
136     uint64_t result = siphash(k_0, k_1, buf, 15);
137     uint64_t expected = 0xa129ca6149be45e5;
138
139     return 0;
140 }
```

C++ 自定义哈希函数

```
1  struct TM_hash {
2    // 32bit integer hash by T. Mueller
3    constexpr std::size_t
4    operator () (std::uint32_t k) const noexcept {
5      k = ((k >> 16) ^ k) * 0x45d9f3b;
6      k = ((k >> 16) ^ k) * 0x45d9f3b;
7      k = ((k >> 16) ^ k);
8      return k;
9    }
10 };
11
12 std::unordered_set<std::uint32_t,TM_hash> s;
```

最长不增子序列

```
1  // 最长不增子序列，动态规划+二分，O(NlogN)
2  // 若要求最长不减子序列，将数组中所有元素取负即可
3  // 若要求最长递减子序列（即，不能出现相等的情况），在调用二分查找时将
       val + 1即可
4  // 注意最长不增子序列的长度为dp中的最大值
5  #include <vector>
6
7  #define MY_INFINITY (1LL << 60)
8
9  typedef long long i64;
10 i64 get_length(std::vector<i64> const &list, i64 left, i64 right, i64
       val);
11 std::vector<i64> calc_dp(std::vector<i64> const &num_list);
12
13 std::vector<i64> calc_dp(std::vector<i64> const &num_list) {
14     std::vector<i64> dp;
15     dp.resize(num_list.size(), 0);
16
17     std::vector<i64> max_last_element_by_length;
18
```

```
    max_last_element_by_length.resize(num_list.size() + 1, -MY_INFINITY
        );

    max_last_element_by_length[0] = MY_INFINITY;

    dp[0] = 1;
    max_last_element_by_length[1] = num_list[0];

    for (size_t i = 1; i < num_list.size(); i++) {
        i64 prev_max_length =
            get_length(max_last_element_by_length, 1, (i64)i, num_list[
                i]);

        dp[i] = prev_max_length + 1;

        if (max_last_element_by_length[(size_t)dp[i]] < num_list[i]) {
            max_last_element_by_length[(size_t)dp[i]] = num_list[i];
        }
    }

    return dp;
}

i64 get_length(std::vector<i64> const &list, i64 left, i64 right, i64
    val) {
    while (left <= right) {
        i64 mid = (left + right) / 2;

        if (list[(size_t)mid] < val) {
            right = mid - 1;
        } else if (list[(size_t)mid] > val) {
            left = mid + 1;
        } else {
            left = mid + 1;
        }
    }

    return right;
}
```

并查集

```
#include <cstddef>
#include <numeric>
#include <vector>

using std::size_t;

struct DisjointSet {
```

```
    std::vector<size_t> set;
    std::vector<size_t> set_size;

    explicit DisjointSet(size_t size) {
        set.resize(size);
        set_size.resize(size, 1);
        std::iota(std::begin(set), std::end(set), 0);
    }

    size_t find(size_t x) {
        std::vector<size_t> path(16);

        while (set[x] != x) {
            path.push_back(x);
            x = set[x];
        };

        for (size_t node : path) {
            set[node] = x;
        }

        return x;
    }

    void unite(size_t x, size_t y) {
        size_t x_root = find(x);
        size_t y_root = find(y);

        if (x_root == y_root) {
            return;
        }

        if (set_size[x_root] < set_size[y_root]) {
            set[x_root] = y_root;
            set_size[y_root] += set_size[x_root];
        } else {
            set[y_root] = x_root;
            set_size[x_root] += set_size[y_root];
        }
    }
};
```

最近公共祖先

```
#include <array>
#include <cstdio>
#include <utility>
#include <vector>

```

```cpp
class LCA {
  public:
    std::vector<std::array<size_t, 32>> parent_info;
    std::vector<size_t> node_depth;

  private:
    std::vector<std::vector<size_t>> const &tree;
    size_t const tree_root;

  public:
    explicit LCA(std::vector<std::vector<size_t>> const &tree, size_t root)
        : tree{tree}, tree_root{root} {
        parent_info.resize(tree.size());
        node_depth.resize(tree.size());

        dfs(tree_root, 0);
    }

  public:
    size_t lca(size_t u, size_t v) {
        if (node_depth[u] > node_depth[v]) {
            std::swap(u, v);
        }

        size_t depth_delta = node_depth[v] - node_depth[u];

        size_t i = 0;

        while (depth_delta > 0) {
            size_t digit = depth_delta & 1;
            depth_delta >>= 1;

            if (digit == 1) {
                v = parent_info[v][i];
            }

            i++;
        }
        while (u != v) {
            for (i = 0; i < 32; i++) {
                size_t idx = 31 - i;

                if (parent_info[u][idx] != parent_info[v][idx]) {
                    u = parent_info[u][idx];
                    v = parent_info[v][idx];

                    break;
                }
            }
        }
```

```
56
57          return u;
58      }
59
60    private:
61      void dfs(size_t root, size_t depth) {
62          node_depth[root] = depth;
63
64          for (size_t child_node : tree[root]) {
65              parent_info[child_node][0] = root;
66
67              for (size_t i = 1; i < 32; i++) {
68                  size_t p = parent_info[child_node][i - 1];
69
70                  // 2 ^ i = 2 ^ (i - 1) + 2 ^ (i - 1)
71                  parent_info[child_node][i] = parent_info[p][i - 1];
72              }
73
74              dfs(child_node, depth + 1);
75          }
76      }
77  };
```

图

DFS

```
1   #include <cstdio>
2   #include <vector>
3
4   size_t dfs_time = 0;
5   std::vector<std::vector<size_t>> graph;
6
7   std::vector<int> visited;
8   std::vector<size_t> parent;
9   std::vector<size_t> discover_time;
10  std::vector<size_t> finish_time;
11
12  void dfs_visit(size_t source_node) {
13      dfs_time++;
14      discover_time[source_node] = dfs_time;
15      visited[source_node] = 1;
16
17      for (size_t adjacent_node : graph[source_node]) {
18          if (visited[adjacent_node] == 0) {
19              parent[adjacent_node] = source_node;
20              discover_time[adjacent_node] = dfs_time;
21              dfs_visit(adjacent_node);
```

```
        }
    }

    dfs_time++;
    finish_time[source_node] = dfs_time;
}

void dfs_graph() {
    dfs_time = 0;

    visited.assign(graph.size(), 0);
    parent.assign(graph.size(), 0);
    discover_time.assign(graph.size(), 0);
    finish_time.assign(graph.size(), 0);

    for (size_t i = 1; i < graph.size(); i++) {
        if (visited[i] == 0) {
            dfs_visit(i);
        }
    }
}

int main(void) {
    size_t n, m;
    n = m = 0;
    std::scanf("%zu%zu", &n, &m);

    graph.resize(n + 1);

    for (size_t i = 0; i < m; i++) {
        size_t from_node, to_node;
        from_node = to_node = 0;

        std::scanf("%zu%zu", &from_node, &to_node);

        graph[from_node].push_back(to_node);
    }

    dfs_graph();

    for (size_t i = 1; i <= n; i++) {
        std::printf("%zu ", parent[i]);
    }

    std::putchar('\n');

    for (size_t i = 1; i <= n; i++) {
        std::printf("%zu ", discover_time[i]);
    }

    std::putchar('\n');
```

```

    for (size_t i = 1; i <= n; i++) {
        std::printf("%zu ", finish_time[i]);
    }

    std::putchar('\n');

    return 0;
}
```

拓扑排序

```cpp
#include <queue>
#include <vector>

struct TopoResult {
    int result_code;
    std::vector<size_t> result;
};

// 0 下标
// Sorted sequence cannot be
// determined, 表示拓扑排序不唯一（条件：任意时刻入度为0的顶点的集合的元素个数大于1）

TopoResult topo_sort(std::vector<std::vector<size_t>> const &graph) {
    // 0 -> Sorted sequence determined
    // 1 -> Inconsistency found
    // 2 -> Sorted sequence cannot be determined.
    int result_code = 0;
    std::vector<size_t> result;
    result.reserve(graph.size());

    std::vector<size_t> in_count;
    in_count.resize(graph.size(), 0);

    for (std::vector<size_t> const &out_list : graph) {
        for (size_t out_node : out_list) {
            in_count[out_node]++;
        }
    }

    std::queue<size_t> node_queue;

    for (size_t i = 0; i < graph.size(); i++) {
        if (in_count[i] == 0) {
            node_queue.push(i);
        }
    }
```

```
37
38      if (node_queue.size() > 1) {
39          result_code = 2;
40      }
41
42      while (!node_queue.empty()) {
43          if (node_queue.size() > 1) {
44              result_code = 2;
45          }
46          size_t current_node = node_queue.front();
47          node_queue.pop();
48
49          result.push_back(current_node);
50
51          for (size_t other_node : graph[current_node]) {
52              if (in_count[other_node] == 1) {
53                  node_queue.push(other_node);
54              }
55
56              in_count[other_node]--;
57          }
58      }
59
60      if (result.size() < graph.size()) {
61          result_code = 1;
62      }
63
64      TopoResult r = {result_code, result};
65
66      return r;
67  }
```

DFS 拓扑排序

```
1   #include <algorithm>
2   #include <cstdio>
3   #include <vector>
4
5   size_t dfs_time = 0;
6   std::vector<std::vector<size_t>> graph;
7
8   std::vector<int> visited;
9   std::vector<size_t> parent;
10  std::vector<size_t> discover_time;
11  std::vector<size_t> finish_time;
12  std::vector<size_t> topo_list;
13
14  void dfs_visit(size_t source_node) {
15      dfs_time++;
```

```
16      discover_time[source_node] = dfs_time;
17      visited[source_node] = 1;
18
19      for (size_t adjacent_node : graph[source_node]) {
20          if (visited[adjacent_node] == 0) {
21              parent[adjacent_node] = source_node;
22              discover_time[adjacent_node] = dfs_time;
23              dfs_visit(adjacent_node);
24          }
25      }
26
27      dfs_time++;
28      finish_time[source_node] = dfs_time;
29
30      topo_list.push_back(source_node);
31  }
32
33  void dfs_graph() {
34      dfs_time = 0;
35
36      visited.assign(graph.size(), 0);
37      parent.assign(graph.size(), 0);
38      discover_time.assign(graph.size(), 0);
39      finish_time.assign(graph.size(), 0);
40      topo_list.resize(0);
41
42      for (size_t i = 1; i < graph.size(); i++) {
43          if (visited[i] == 0) {
44              dfs_visit(i);
45          }
46      }
47
48      std::reverse(std::begin(topo_list), std::end(topo_list));
49  }
50
51  int main(void) {
52      size_t N = 0;
53
54      std::scanf("%zu", &N);
55
56      graph.resize(N + 1);
57
58      for (size_t i = 1; i <= N; i++) {
59          size_t a = 0;
60
61          while (std::scanf("%zu", &a) == 1) {
62              if (a == 0) {
63                  break;
64              }
65
66              graph[i].push_back(a);
```

```
        }
    }

    dfs_graph();

    for (size_t node : topo_list) {
        std::printf("%zu ", node);
    }

    std::putchar('\n');

    return 0;
}
```

DFS 强连通分量

```
#include <algorithm>
#include <cstdio>
#include <vector>

size_t dfs_time;
std::vector<int> visited;
std::vector<size_t> finish_order;
std::vector<std::vector<size_t>> graph;
std::vector<std::vector<size_t>> reversed_graph;
std::vector<std::vector<size_t>> scc_by_node;

void dfs_pass1();
void dfs_visit_pass1(size_t source_node);
void dfs_pass2();
void dfs_visit_pass2(size_t source_node, size_t tree_root);
void generate_reverse_graph();

int main(void) {
    size_t n, m;
    n = m = 0;

    std::scanf("%zu%zu", &n, &m);

    graph.resize(n + 1);

    for (size_t i = 0; i < m; i++) {
        size_t from_node, to_node;
        from_node = to_node = 0;

        std::scanf("%zu%zu", &from_node, &to_node);

        graph[from_node].push_back(to_node);
    }
```

```
34
35      generate_reverse_graph();
36      dfs_pass1();
37      dfs_pass2();
38
39      for (size_t i = 1; i <= n; i++) {
40          for (size_t scc_node : scc_by_node[i]) {
41              std::printf("%zu ", scc_node);
42          }
43          if (!scc_by_node[i].empty()) {
44              std::putchar('\n');
45          }
46      }
47
48      return 0;
49  }
50
51  void dfs_pass1() {
52      dfs_time = 0;
53      visited.assign(graph.size(), 0);
54      finish_order.clear();
55
56      for (size_t i = 0; i < graph.size(); i++) {
57          if (visited[i] == 0) {
58              dfs_visit_pass1(i);
59          }
60      }
61
62      std::reverse(std::begin(finish_order), std::end(finish_order));
63  }
64
65  void dfs_visit_pass1(size_t source_node) {
66      dfs_time++;
67      visited[source_node] = 1;
68
69      for (size_t adjacent_node : graph[source_node]) {
70          if (visited[adjacent_node] == 0) {
71              dfs_visit_pass1(adjacent_node);
72          }
73      }
74
75      dfs_time++;
76
77      finish_order.push_back(source_node);
78  }
79
80  void dfs_pass2() {
81      visited.assign(graph.size(), 0);
82      scc_by_node.resize(graph.size());
83
84      for (size_t node : finish_order) {
```

```
        if (visited[node] == 0) {
            dfs_visit_pass2(node, node);
        }
    }
}

void dfs_visit_pass2(size_t source_node, size_t tree_root) {
    visited[source_node] = 1;
    scc_by_node[tree_root].push_back(source_node);

    for (size_t adjacent_node : reversed_graph[source_node]) {
        if (visited[adjacent_node] == 0) {
            dfs_visit_pass2(adjacent_node, tree_root);
        }
    }
}

void generate_reverse_graph() {
    reversed_graph.resize(graph.size());

    for (size_t i = 1; i < graph.size(); i++) {
        for (size_t adjacent_node : graph[i]) {
            reversed_graph[adjacent_node].push_back(i);
        }
    }
}
```

最小生成树

Kruskal

```
#include <algorithm>
#include <numeric>
#include <vector>

struct DisjointSet {
    std::vector<size_t> set;
    std::vector<size_t> set_size;

    explicit DisjointSet(size_t size) {
        set.resize(size);
        set_size.resize(size, 1);
        std::iota(std::begin(set), std::end(set), 0);
    }

    size_t find(size_t x) {
        std::vector<size_t> path(16);

        while (set[x] != x) {
            path.push_back(x);
            x = set[x];
```

```
        };

        for (size_t node : path) {
            set[node] = x;
        }

        return x;
    }

    void unite(size_t x, size_t y) {
        size_t x_root = find(x);
        size_t y_root = find(y);

        if (x_root == y_root) {
            return;
        }

        if (set_size[x_root] < set_size[y_root]) {
            set[x_root] = y_root;
            set_size[y_root] += set_size[x_root];
        } else {
            set[y_root] = x_root;
            set_size[x_root] += set_size[y_root];
        }
    }
};

// 1 下标

struct Edge {
    size_t from;
    size_t to;
    int weight;

    explicit constexpr Edge(size_t from_, size_t to_, int weight_)
        : from{from_}, to{to_}, weight{weight_} {};
};

// 副作用：传入的edge_list将被排序

std::vector<Edge> Kruskal(std::vector<Edge> &edge_list, size_t
    node_count) {
    std::sort(std::begin(edge_list), std::end(edge_list),
              [](Edge const &a, Edge const &b) { return a.weight < b.
                  weight; });

    DisjointSet set{node_count + 1};

    std::vector<Edge> result;

    for (Edge const &edge : edge_list) {
```

```
        size_t from = edge.from;
        size_t to = edge.to;

        size_t from_root = set.find(from);
        size_t to_root = set.find(to);

        if (from_root != to_root) {
            result.push_back(edge);
            set.unite(from_root, to_root);
        }

        if (result.size() == (node_count - 1)) {
            break;
        }
    }

    return result;
}
```

Prim

```
#include <queue>
#include <vector>

typedef long long i64;

// 顶点1下标, 边1下标

struct Edge {
    size_t from;
    size_t to;
    i64 weight;

    explicit constexpr Edge(size_t from_, size_t to_, i64 weight_)
        : from{from_}, to{to_}, weight{weight_} {};
};

struct NodeInfo {
    size_t node;
    size_t parent;
    size_t min_weight_edge_id;
    i64 min_weight_to_tree;

    explicit constexpr NodeInfo(size_t node_, size_t parent_,
                                size_t min_weight_edge_id_, i64 weight_
                                    )
        : node{node_}, parent{parent_}, min_weight_edge_id{
            min_weight_edge_id_},
          min_weight_to_tree{weight_} {};

    friend bool operator<(NodeInfo const &left, NodeInfo const &right)
```

```
        {
29          return left.min_weight_to_tree > right.min_weight_to_tree;
30      }
31  };
32
33  std::vector<Edge> Prim(std::vector<std::vector<Edge>> const &graph) {
34      std::vector<int> visited;
35      visited.resize(graph.size(), 0);
36
37      std::vector<Edge> result;
38      result.reserve(graph.size() - 1);
39
40      std::priority_queue<NodeInfo> node_queue;
41
42      visited[1] = 1;
43
44      for (size_t i = 0; i < graph[1].size(); i++) {
45          size_t target_node = graph[1][i].to;
46          i64 weight = graph[1][i].weight;
47          node_queue.emplace(target_node, 1, i, weight);
48      }
49
50      while (!node_queue.empty()) {
51          NodeInfo next_node_info = node_queue.top();
52          node_queue.pop();
53
54          if (visited[next_node_info.node] == 0) {
55              visited[next_node_info.node] = 1;
56
57              result.push_back(graph[next_node_info.parent]
58                                    [next_node_info.min_weight_edge_id]);
59
60              for (size_t i = 0; i < graph[next_node_info.node].size(); i
                    ++) {
61                  size_t target_node = graph[next_node_info.node][i].to;
62                  i64 weight = graph[next_node_info.node][i].weight;
63                  node_queue.emplace(target_node, next_node_info.node, i,
                        weight);
64              }
65          }
66
67          if (result.size() == graph.size() - 1) {
68              break;
69          }
70      }
71
72      return result;
73  }
```

最短路

BF

```cpp
#include <vector>

#define MY_INFINITY (1LL << 61)

typedef long long i64;

struct Edge {
    size_t to;
    i64 weight;

    explicit constexpr Edge(size_t to_, i64 weight_)
        : to{to_}, weight{weight_} {};
};

struct BFResult {
    bool valid;
    std::vector<i64> shortest_distance;
    std::vector<size_t> parent;
};

BFResult BF(std::vector<std::vector<Edge>> const &graph, size_t source_node) {
    bool valid = true;
    std::vector<i64> shortest_distance;
    std::vector<size_t> parent;

    shortest_distance.resize(graph.size(), MY_INFINITY);
    parent.resize(graph.size(), 0);

    shortest_distance[source_node] = 0;

    for (size_t i = 0; i < graph.size() - 1; i++) {
        for (size_t from_node = 1; from_node < graph.size(); from_node++) {
            for (Edge const &edge : graph[from_node]) {
                size_t to_node = edge.to;

                if (shortest_distance[from_node] + edge.weight <
                    shortest_distance[to_node]) {
                    shortest_distance[to_node] =
                        shortest_distance[from_node] + edge.weight;
                    parent[to_node] = from_node;
                }
            }
        }
    }

    for (size_t from_node = 1; from_node < graph.size(); from_node++) {
        for (Edge const &edge : graph[from_node]) {
```

```
            if (shortest_distance[from_node] + edge.weight <
                shortest_distance[edge.to]) {
                valid = false;
                break;
            }
        }

        if (!valid) {
            break;
        }
    }

    return {valid, shortest_distance, parent};
}
```

有向无环图

```
#include <queue>
#include <vector>

#define MY_INFINITY (1LL << 61)

typedef long long i64;

struct Edge {
    size_t to;
    i64 weight;

    explicit constexpr Edge(size_t to_, i64 weight_)
        : to{to_}, weight{weight_} {};
};

struct TopoResult {
    int result_code;
    std::vector<size_t> result;
};

struct DistanceResult {
    std::vector<i64> shortest_distance;
    std::vector<size_t> parent;
};

TopoResult topo_sort(std::vector<std::vector<Edge>> const &graph);
DistanceResult DAGShortestPath(std::vector<std::vector<Edge>> const &
    graph,
                               size_t source_node);

// Sorted sequence cannot be
// determined, 表示拓扑排序不唯一（条件：任意时刻入度为0的顶点的集合的
    元素个数大于1）

```

```cpp
TopoResult topo_sort(std::vector<std::vector<Edge>> const &graph) {
    // 0 -> Sorted sequence determined
    // 1 -> Inconsistency found
    // 2 -> Sorted sequence cannot be determined.
    int result_code = 0;
    std::vector<size_t> result;
    result.reserve(graph.size());

    std::vector<size_t> in_count;
    in_count.resize(graph.size(), 0);

    for (std::vector<Edge> const &out_list : graph) {
        for (Edge const &edge : out_list) {
            in_count[edge.to]++;
        }
    }

    std::queue<size_t> node_queue;

    for (size_t i = 0; i < graph.size(); i++) {
        if (in_count[i] == 0) {
            node_queue.push(i);
        }
    }

    if (node_queue.size() > 1) {
        result_code = 2;
    }

    while (!node_queue.empty()) {
        if (node_queue.size() > 1) {
            result_code = 2;
        }
        size_t current_node = node_queue.front();
        node_queue.pop();

        result.push_back(current_node);

        for (Edge const &edge : graph[current_node]) {
            size_t other_node = edge.to;

            if (in_count[other_node] == 1) {
                node_queue.push(other_node);
            }

            in_count[other_node]--;
        }
    }

    if (result.size() < graph.size()) {
        result_code = 1;
```

```
    }

    TopoResult r = {result_code, result};

    return r;
}

DistanceResult DAGShortestPath(std::vector<std::vector<Edge>> const &
    graph,
                               size_t source_node) {
    std::vector<i64> distance;
    std::vector<size_t> parent;

    TopoResult topo_result = topo_sort(graph);

    std::vector<size_t> topo_list = topo_result.result;

    distance.resize(graph.size(), MY_INFINITY);
    parent.resize(graph.size(), 0);

    distance[source_node] = 0;

    for (size_t from_node : topo_list) {
        for (Edge const &edge : graph[from_node]) {
            size_t to_node = edge.to;

            if (distance[from_node] + edge.weight < distance[to_node])
                {
                distance[to_node] = distance[from_node] + edge.weight;
                parent[to_node] = from_node;
            }
        }
    }

    return {distance, parent};
}
```

Dijkstra

```
#include <cstddef>
#include <queue>
#include <vector>

#define MY_INFINITY (1LL << 60)

typedef long long i64;

struct Edge {
    size_t to;
    i64 weight;

```

```
13      explicit constexpr Edge(size_t to_, i64 weight_)
14          : to{to_}, weight{weight_} {}
15  };
16
17  struct NodeInfo {
18      size_t node;
19      i64 distance;
20
21      friend bool operator<(NodeInfo const &left, NodeInfo const &right)
            {
22          return left.distance > right.distance;
23      }
24
25      explicit constexpr NodeInfo(size_t node_, i64 distance_)
26          : node{node_}, distance{distance_} {}
27  };
28
29  std::vector<i64> dijkstra(std::vector<std::vector<Edge>> const &
        node_to_edges,
30                            size_t source_node) {
31      std::vector<i64> result;
32      std::vector<int> visited;
33
34      result.resize(node_to_edges.size(), MY_INFINITY);
35      visited.resize(node_to_edges.size(), 0);
36
37      std::priority_queue<NodeInfo> node_queue;
38
39      result[source_node] = 0;
40      node_queue.emplace(source_node, 0);
41
42      while (!node_queue.empty()) {
43          NodeInfo current_node_info = node_queue.top();
44
45          node_queue.pop();
46
47          if (visited[current_node_info.node] == 1) {
48              continue;
49          }
50
51          for (Edge const &edge : node_to_edges[current_node_info.node])
                {
52              size_t other_node = edge.to;
53
54              if (result[other_node] >
55                  result[current_node_info.node] + edge.weight) {
56                  result[other_node] =
57                      result[current_node_info.node] + edge.weight;
58                  node_queue.emplace(other_node, result[other_node]);
59              }
60          }
```

```
61
62          visited[current_node_info.node] = 1;
63      }
64
65      return result;
66  }
```

最大流

EK

```
1   // O(V(E^2))
2   // 处理了反平行边、重边、自环
3   // 反平行边：新建节点
4   // 重边：容量相加
5   // 自环：直接移除
6
7   #include <cstdio>
8   #include <cstring>
9   #include <queue>
10  #include <vector>
11
12  #define MAX_NODE 2005
13  #define INFINITY (1LL << 60)
14
15  typedef long long i64;
16
17  i64 residual_graph[MAX_NODE][MAX_NODE];
18
19  struct Edge {
20      size_t to;
21      i64 weight;
22
23      explicit constexpr Edge(size_t to_, i64 weight_)
24          : to{to_}, weight{weight_} {}
25  };
26
27  class EK {
28    public:
29      size_t node_count;
30
31    private:
32      std::vector<std::vector<size_t>> node_to_edges;
33
34    public:
35      explicit EK(std::vector<std::vector<Edge>> const &graph) {
36          std::memset(residual_graph, 0, sizeof(residual_graph));
37          node_count = graph.size() - 1;
38          node_to_edges.resize(graph.size());
39
40          for (size_t from_node = 0; from_node < graph.size(); from_node
```

```
            ++) {
            for (Edge const &edge : graph[from_node]) {
                size_t to_node = edge.to;
                i64 weight = edge.weight;

                // antiparallel edge exists
                // parallel edges are allowed
                if (residual_graph[to_node][from_node] != 0) {
                    node_count++;

                    residual_graph[from_node][node_count] += weight;
                    residual_graph[node_count][to_node] += weight;

                    if (node_count > node_to_edges.size() - 1) {
                        node_to_edges.resize(2 * node_count);
                    }

                    // 注意须在node_to_edges（对应残余图）中建立反向边
                    node_to_edges[from_node].push_back(node_count);
                    node_to_edges[node_count].push_back(from_node);
                    node_to_edges[node_count].push_back(to_node);
                    node_to_edges[to_node].push_back(node_count);
                } else {
                    residual_graph[from_node][to_node] += weight;
                    node_to_edges[from_node].push_back(to_node);
                    node_to_edges[to_node].push_back(from_node);
                }
            }
        }
        // remove self loop

        for (size_t i = 0; i <= node_count; i++) {
            residual_graph[i][i] = 0;
        }
    }

    i64 max_flow(size_t source_node, size_t target_node) {
        i64 max_flow = 0;

        while (true) {
            i64 augment = rg_bfs(source_node, target_node);

            // std::printf("augment: %lld\n", augment);

            if (augment == 0) {
                break;
            }

            max_flow += augment;
        }

```

```
91          return max_flow;
92      }
93
94    private:
95      i64 rg_bfs(size_t from_node, size_t to_node) {
96          std::vector<int> visited;
97
98          visited.resize(node_count + 1, 0);
99
100         std::queue<size_t> node_queue;
101         std::vector<size_t> parent;
102
103         parent.resize(node_count + 1, 0);
104
105         node_queue.push(from_node);
106
107         visited[from_node] = 1;
108
109         while (!node_queue.empty()) {
110             size_t current_node = node_queue.front();
111
112             node_queue.pop();
113
114             if (current_node == to_node) {
115                 break;
116             }
117
118             for (size_t next_node : node_to_edges[current_node]) {
119                 if (residual_graph[current_node][next_node] > 0) {
120                     if (visited[next_node] == 0) {
121                         visited[next_node] = 1;
122                         parent[next_node] = current_node;
123                         node_queue.push(next_node);
124                     }
125                 }
126             }
127         }
128
129         if (!visited[to_node]) {
130             return 0;
131         }
132
133         size_t current_node = to_node;
134         i64 augment = INFINITY;
135
136         while (current_node != from_node) {
137             augment = std::min(
138                 augment, residual_graph[parent[current_node]][
                        current_node]);
139             current_node = parent[current_node];
140         }
```

```
141
142         current_node = to_node;
143
144         while (current_node != from_node) {
145             residual_graph[parent[current_node]][current_node] -=
                augment;
146             residual_graph[current_node][parent[current_node]] +=
                augment;
147
148             current_node = parent[current_node];
149         }
150
151         return augment;
152     }
153 };
154
155 int main() {
156     int T = 0;
157     std::scanf("%d", &T);
158
159     for (int id = 0; id < T; id++) {
160
161         size_t node_count, edge_count, source_node, target_node;
162
163         std::scanf("%zu%zu%zu%zu", &node_count, &edge_count, &
            source_node,
164                    &target_node);
165
166         std::vector<std::vector<Edge>> graph;
167
168         graph.resize(node_count + 1);
169
170         for (size_t i = 0; i < edge_count; i++) {
171             size_t from_node, to_node;
172             from_node = to_node = 0;
173             i64 weight = 0;
174
175             std::scanf("%zu%zu%lld", &from_node, &to_node, &weight);
176
177             graph[from_node].emplace_back(to_node, weight);
178         }
179
180         EK ek{graph};
181
182         std::printf("%lld\n", ek.max_flow(source_node, target_node));
183     }
184 }
```

Dinic

```
1 // O((V^2)E)
```

```cpp
// 处理了反平行边、重边、自环
// 反平行边：新建节点
// 重边：容量相加
// 自环：直接移除

#include <cstdio>
#include <cstring>
#include <queue>
#include <vector>

#define MAX_NODE 2005
#define INFINITY (1LL << 60)

typedef long long i64;

i64 residual_graph[MAX_NODE][MAX_NODE];

struct Edge {
    size_t to;
    i64 weight;

    explicit constexpr Edge(size_t to_, i64 weight_)
        : to{to_}, weight{weight_} {}
};

class Dinic {
  public:
    size_t node_count;

  private:
    std::vector<std::vector<size_t>> node_to_edges;

  public:
    explicit Dinic(std::vector<std::vector<Edge>> const &graph) {
        std::memset(residual_graph, 0, sizeof(residual_graph));
        node_count = graph.size() - 1;
        node_to_edges.resize(graph.size());

        for (size_t from_node = 0; from_node < graph.size(); from_node
            ++) {
            for (Edge const &edge : graph[from_node]) {
                size_t to_node = edge.to;
                i64 weight = edge.weight;

                // antiparallel edge exists
                // parallel edges are allowed
                if (residual_graph[to_node][from_node] != 0) {
                    node_count++;

                    residual_graph[from_node][node_count] += weight;
                    residual_graph[node_count][to_node] += weight;
```

```
52
53                      if (node_count > node_to_edges.size() - 1) {
54                          node_to_edges.resize(2 * node_count);
55                      }
56
57                      // 注意须在node_to_edges（对应残余图）中建立反向边
58                      node_to_edges[from_node].push_back(node_count);
59                      node_to_edges[node_count].push_back(from_node);
60                      node_to_edges[node_count].push_back(to_node);
61                      node_to_edges[to_node].push_back(node_count);
62                  } else {
63                      residual_graph[from_node][to_node] += weight;
64                      node_to_edges[from_node].push_back(to_node);
65                      node_to_edges[to_node].push_back(from_node);
66                  }
67              }
68          }
69          // remove self loop
70
71          for (size_t i = 0; i <= node_count; i++) {
72              residual_graph[i][i] = 0;
73          }
74      }
75
76      i64 max_flow(size_t source_node, size_t target_node) {
77          i64 max_flow = 0;
78
79          while (true) {
80              std::vector<size_t> node_to_depth =
81                  rg_bfs(source_node, target_node);
82
83              // 汇点已经不可达，不可能再有增广路径
84              if (node_to_depth[target_node] == 0) {
85                  break;
86              }
87
88              i64 augment =
89                  dfs(source_node, target_node, INFINITY, node_to_depth);
90
91              if (augment == 0) {
92                  break;
93              }
94
95              max_flow += augment;
96          }
97
98          return max_flow;
99      }
100
101   private:
102     // BFS分层，返回每个节点的层数
```

```
    std::vector<size_t> rg_bfs(size_t from_node, size_t to_node) {
        std::vector<int> visited;

        visited.resize(node_count + 1, 0);

        std::queue<size_t> node_queue;
        std::vector<size_t> parent;
        std::vector<size_t> depth;

        parent.resize(node_count + 1, 0);
        depth.resize(node_count + 1, 0);

        node_queue.push(from_node);
        depth[from_node] = 1;
        visited[from_node] = 1;

        while (!node_queue.empty()) {
            size_t current_node = node_queue.front();

            node_queue.pop();

            if (current_node == to_node) {
                break;
            }

            for (size_t next_node : node_to_edges[current_node]) {
                if (residual_graph[current_node][next_node] > 0) {
                    if (visited[next_node] == 0) {
                        depth[next_node] = depth[current_node] + 1;
                        visited[next_node] = 1;
                        parent[next_node] = current_node;
                        node_queue.push(next_node);
                    }
                }
            }
        }

        return depth;
    }

    // in_flow: 尝试输入该节点的流量, 对于入点, 为INFINITY
    // 返回: 实际成功输入的流量
    i64 dfs(size_t from_node, size_t to_node, i64 in_flow,
            std::vector<size_t> &depth) {
        if ((from_node == to_node) || (in_flow == 0)) {
            return in_flow;
        }

        // 还需要从该节点的出边分配的流量
        i64 remain_flow = in_flow;

```

```
154             for (size_t next_node : node_to_edges[from_node]) {
155                 // 仅考虑下一层的节点
156                 if (depth[next_node] == depth[from_node] + 1) {
157                     if (residual_graph[from_node][next_node] > 0) {
158                         // 对于每一条出边，尝试分配尽可能多的流量
159                         i64 to_allocate_flow = std::min(
160                             remain_flow, residual_graph[from_node][
                                next_node]);
161 
162                         // 实际成功分配的流量
163                         i64 allocated_flow =
164                             dfs(next_node, to_node, to_allocate_flow, depth
                                );
165 
166                         // 下层节点已经枯竭，在之后的所有操作中都不再尝试向
                            该节点分配流量
167                         if (allocated_flow == 0) {
168                             depth[next_node] = 0;
169                         }
170 
171                         remain_flow -= allocated_flow;
172 
173                         residual_graph[from_node][next_node] -=
                            allocated_flow;
174                         residual_graph[next_node][from_node] +=
                            allocated_flow;
175                     }
176                 }
177 
178                 if (remain_flow == 0) {
179                     break;
180                 }
181             }
182 
183             return (in_flow - remain_flow);
184         }
185     };
186 
187     int main() {
188         int T = 0;
189         std::scanf("%d", &T);
190 
191         for (int id = 0; id < T; id++) {
192 
193             size_t node_count, edge_count, source_node, target_node;
194 
195             std::scanf("%zu%zu%zu%zu", &node_count, &edge_count, &
                source_node,
196                        &target_node);
197 
198             std::vector<std::vector<Edge>> graph;
```

```
199
200         graph.resize(node_count + 1);
201
202         for (size_t i = 0; i < edge_count; i++) {
203             size_t from_node, to_node;
204             from_node = to_node = 0;
205             i64 weight = 0;
206
207             std::scanf("%zu%zu%lld", &from_node, &to_node, &weight);
208
209             graph[from_node].emplace_back(to_node, weight);
210         }
211
212         Dinic dinic{graph};
213
214         std::printf("%lld\n", dinic.max_flow(source_node, target_node))
                ;
215     }
216 }
```

FFT

```
1  #include <cmath>
2  #include <complex>
3  #include <cstdio>
4  #include <vector>
5
6  double const PI = std::acos(-1);
7
8  using Complex = std::complex<double>; // STL complex
9
10 constexpr int MAX_N = 1 << 20;
11
12 int rev[MAX_N];
13
14 void change(Complex y[], int len);
15 void fft(Complex y[], int len, int on);
16
17 int main(void) {
18     std::vector<double> v{1, 1, -1, 2, 1, 0, -1, 1};
19     std::vector<Complex> cv;
20     cv.resize(v.size());
21
22     for (size_t i = 0; i < v.size(); i++) {
23         cv[i].real(v[i]);
24         cv[i].imag(0);
25     }
26
27     fft(cv.data(), cv.size(), 1);
```

```
28
29      for (size_t i = 0; i < v.size(); i++) {
30          std::printf("%.5f + %.5fi\n", cv[i].real(), cv[i].imag());
31      }
32  }
33
34  // 同样需要保证 len 是 2 的幂
35  // 记 rev[i] 为 i 翻转后的值
36  void change(Complex y[], int len) {
37      for (int i = 0; i < len; ++i) {
38          rev[i] = rev[i >> 1] >> 1;
39          if (i & 1) { // 如果最后一位是 1，则翻转成 len/2
40              rev[i] |= len >> 1;
41          }
42      }
43      for (int i = 0; i < len; ++i) {
44          if (i < rev[i]) { // 保证每对数只翻转一次
45              swap(y[i], y[rev[i]]);
46          }
47      }
48      return;
49  }
50
51  /*
52   * 做 FFT
53   * len 必须是 2^k 形式
54   * on == 1 时是 DFT，on == -1 时是 IDFT
55   */
56  void fft(Complex y[], int len, int on) {
57      // 位逆序置换
58      change(y, len);
59      // 模拟合并过程，一开始，从长度为一合并到长度为二，一直合并到长度为
            len。
60      for (int h = 2; h <= len; h <<= 1) {
61          // wn：当前单位复根的间隔：w^1_h
62          Complex wn(cos(2 * PI / h), sin(on * 2 * PI / h));
63          // 合并，共 len / h 次。
64          for (int j = 0; j < len; j += h) {
65              // 计算当前单位复根，一开始是 1 = w^0_n，之后是以 wn 为间隔
                    递增：
66              // w^1_n
67              // ...
68              Complex w(1, 0);
69              for (int k = j; k < j + h / 2; k++) {
70                  // 左侧部分和右侧是子问题的解
71                  Complex u = y[k];
72                  Complex t = w * y[k + h / 2];
73                  // 这就是把两部分分治的结果加起来
74                  y[k] = u + t;
75                  y[k + h / 2] = u - t;
76                  // 后半个「step」中的ω一定和「前半个」中的成相反数
```

```
                // 「红圈」上的点转一整圈「转回来」，转半圈正好转成相反数
                // 一个数相反数的平方与这个数自身的平方相等
                w = w * wn;
            }
        }
    }
    // 如果是 IDFT，它的逆矩阵的每一个元素不只是原元素取倒数，还要除以长度 len。
    if (on == -1) {
        for (int i = 0; i < len; i++) {
            y[i].real(y[i].real() / len);
            y[i].imag(y[i].imag() / len);
        }
    }
}
```

多项式乘法

```
#include <algorithm>
#include <cmath>
#include <complex>
#include <cstdio>
#include <vector>

#define EPS 1e-6

typedef double number;
typedef long long i64;

double const PI = std::acos(-1);

using Complex = std::complex<number>; // STL complex
void change(Complex y[], int len);
void fft(Complex y[], int len, int on);

class Multiplier {
  private:
    std::vector<Complex> a_coff_list;
    std::vector<Complex> b_coff_list;
    size_t a_len;
    size_t b_len;
    size_t input_len;

  public:
    explicit Multiplier(std::vector<number> const &a_list,
                        std::vector<number> const &b_list) {
        input_len = std::max(a_list.size(), b_list.size());
        size_t len = nextPowerOfTwo(2 * input_len);
```

```
31
32          a_len = a_list.size();
33          b_len = b_list.size();
34
35          a_coff_list.resize(len);
36          b_coff_list.resize(len);
37
38          for (size_t i = 0; i < a_list.size(); i++) {
39              a_coff_list[i].real(a_list[i]);
40          }
41
42          for (size_t i = 0; i < b_list.size(); i++) {
43              b_coff_list[i].real(b_list[i]);
44          }
45      }
46
47      // can only be called once!!!
48      std::vector<number> multiply() {
49          fft(a_coff_list, false);
50          fft(b_coff_list, false);
51
52          std::vector<Complex> c_coff_list;
53          c_coff_list.resize(a_coff_list.size());
54
55          std::transform(std::begin(a_coff_list), std::end(a_coff_list),
56                         std::begin(b_coff_list), std::begin(c_coff_list)
                               ,
57                         [](Complex a, Complex b) { return a * b; });
58
59          fft(c_coff_list, true);
60
61          std::vector<number> result;
62          result.resize(a_len + b_len - 1);
63
64          std::transform(std::begin(c_coff_list),
65                         std::begin(c_coff_list) + (long)(a_len + b_len -
                                1),
66                         std::begin(result), [](Complex a) { return a.
                            real(); });
67
68          return result;
69      }
70
71    private:
72      size_t nextPowerOfTwo(size_t input) {
73          if (input == 0) {
74              return 1;
75          }
76
77          input--;
78
```

```
79          input |= (input >> 1);
80          input |= (input >> 2);
81          input |= (input >> 4);
82          input |= (input >> 8);
83          input |= (input >> 16);
84          input |= (input >> 32);
85
86          return (input + 1);
87      }
88
89      // 同样需要保证 len 是 2 的幂
90      // 记 rev[i] 为 i 翻转后的值
91      void change(std::vector<Complex> &y) {
92          std::vector<size_t> rev;
93          size_t len = y.size();
94          rev.resize(len, 0);
95
96          for (size_t i = 0; i < len; ++i) {
97              rev[i] = rev[i >> 1] >> 1;
98              if (i & 1) { // 如果最后一位是 1，则翻转成 len/2
99                  rev[i] |= len >> 1;
100             }
101         }
102         for (size_t i = 0; i < len; ++i) {
103             if (i < rev[i]) { // 保证每对数只翻转一次
104                 std::swap(y[i], y[rev[i]]);
105             }
106         }
107         return;
108     }
109
110     /*
111      * 做 FFT
112      * len 必须是 2^k 形式
113      * reverse == false 时是 DFT，reverse == true 时是 IDFT
114      */
115     void fft(std::vector<Complex> &y, bool reverse) {
116         size_t len = y.size();
117
118         // 位逆序置换
119
120         change(y);
121
122         int on = 1;
123         if (reverse) {
124             on = -1;
125         }
126
127         // 模拟合并过程，一开始，从长度为一合并到长度为二，一直合并到长
                度为
128         // len。
```

```
129         for (size_t h = 2; h <= len; h <<= 1) {
130             // wn: 当前单位复根的间隔: w^1_h
131             Complex wn(cos(2 * PI / (double)h), sin(on * 2 * PI / (
                double)h));
132             // 合并, 共 len / h 次。
133             for (size_t j = 0; j < len; j += h) {
134                 // 计算当前单位复根, 一开始是 1 = w^0_n, 之后是以 wn
135                 // 为间隔递增: w^1_n
136                 // ...
137                 Complex w(1, 0);
138                 for (size_t k = j; k < j + h / 2; k++) {
139                     // 左侧部分和右侧是子问题的解
140                     Complex u = y[k];
141                     Complex t = w * y[k + h / 2];
142                     // 这就是把两部分分治的结果加起来
143                     y[k] = u + t;
144                     y[k + h / 2] = u - t;
145                     // 后半个 「step」 中的ω一定和 「前半个」 中的成相
                        反数
146                     // 「红圈」上的点转一整圈「转回来」, 转半圈正好转成
                        相反数
147                     // 一个数相反数的平方与这个数自身的平方相等
148                     w = w * wn;
149                 }
150             }
151         }
152         // 如果是 IDFT, 它的逆矩阵的每一个元素不只是原元素取倒数, 还要
            除以长度
153         // len。
154         if (on == -1) {
155             for (size_t i = 0; i < len; i++) {
156                 y[i].real(y[i].real() / (double)len);
157             }
158         }
159     }
160 };
161
162 int main(void) {
163     size_t a_len = 0;
164     size_t b_len = 0;
165     std::scanf("%zu%zu", &a_len, &b_len);
166
167     a_len++;
168     b_len++;
169
170     std::vector<number> a_list;
171     std::vector<number> b_list;
172     a_list.reserve(a_len);
173     b_list.reserve(b_len);
174
175     for (size_t i = 0; i < a_len; i++) {
```

```
        double input = 0;
        std::scanf("%lf", &input);
        a_list.push_back(input);
    }

    for (size_t i = 0; i < b_len; i++) {
        double input = 0;
        std::scanf("%lf", &input);
        b_list.push_back(input);
    }

    Multiplier mul{a_list, b_list};

    std::vector<number> result = mul.multiply();

    for (size_t i = 0; i < result.size(); i++) {
        std::printf("%lld ", (i64)(result[i] + 0.5));
    }

    std::putchar('\n');

    return 0;
}
```

大整数乘法

```
#include <algorithm>
#include <cmath>
#include <complex>
#include <cstdio>
#include <cstring>
#include <vector>

#define BUFFER_LEN 1000005
#define EPS 1e-6

typedef double number;
typedef long long i64;

double const PI = std::acos(-1);

char buffer[BUFFER_LEN] = {0};

using Complex = std::complex<number>; // STL complex
void change(Complex y[], int len);
void fft(Complex y[], int len, int on);
std::vector<number> get_big_int();

class Multiplier {
```

```
24    private:
25      std::vector<Complex> a_coff_list;
26      std::vector<Complex> b_coff_list;
27      size_t a_len;
28      size_t b_len;
29      size_t input_len;
30
31    public:
32      explicit Multiplier(std::vector<number> const &a_list,
33                          std::vector<number> const &b_list) {
34          input_len = std::max(a_list.size(), b_list.size());
35          size_t len = nextPowerOfTwo(2 * input_len);
36
37          a_len = a_list.size();
38          b_len = b_list.size();
39
40          a_coff_list.resize(len);
41          b_coff_list.resize(len);
42
43          for (size_t i = 0; i < a_list.size(); i++) {
44              a_coff_list[i].real(a_list[i]);
45          }
46
47          for (size_t i = 0; i < b_list.size(); i++) {
48              b_coff_list[i].real(b_list[i]);
49          }
50      }
51
52      // can only be called once!!!
53      std::vector<number> multiply() {
54          fft(a_coff_list, false);
55          fft(b_coff_list, false);
56
57          std::vector<Complex> c_coff_list;
58          c_coff_list.resize(a_coff_list.size());
59
60          std::transform(std::begin(a_coff_list), std::end(a_coff_list),
61                         std::begin(b_coff_list), std::begin(c_coff_list)
                               ,
62                         [](Complex a, Complex b) { return a * b; });
63
64          fft(c_coff_list, true);
65
66          std::vector<number> result;
67          result.resize(a_len + b_len - 1);
68
69          std::transform(std::begin(c_coff_list),
70                         std::begin(c_coff_list) + (long)(a_len + b_len -
                                1),
71                         std::begin(result), [](Complex a) { return a.
                            real(); });
```

```
72
73          return result;
74      }
75
76  private:
77      size_t nextPowerOfTwo(size_t input) {
78          if (input == 0) {
79              return 1;
80          }
81
82          input--;
83
84          input |= (input >> 1);
85          input |= (input >> 2);
86          input |= (input >> 4);
87          input |= (input >> 8);
88          input |= (input >> 16);
89          input |= (input >> 32);
90
91          return (input + 1);
92      }
93
94      // 同样需要保证 len 是 2 的幂
95      // 记 rev[i] 为 i 翻转后的值
96      void change(std::vector<Complex> &y) {
97          std::vector<size_t> rev;
98          size_t len = y.size();
99          rev.resize(len, 0);
100
101         for (size_t i = 0; i < len; ++i) {
102             rev[i] = rev[i >> 1] >> 1;
103             if (i & 1) { // 如果最后一位是 1，则翻转成 len/2
104                 rev[i] |= len >> 1;
105             }
106         }
107         for (size_t i = 0; i < len; ++i) {
108             if (i < rev[i]) { // 保证每对数只翻转一次
109                 std::swap(y[i], y[rev[i]]);
110             }
111         }
112         return;
113     }
114
115     /*
116      * 做 FFT
117      * len 必须是 2^k 形式
118      * reverse == false 时是 DFT，reverse == true 时是 IDFT
119      */
120     void fft(std::vector<Complex> &y, bool reverse) {
121         size_t len = y.size();
122
```

```
123         // 位逆序置换
124
125         change(y);
126
127         int on = 1;
128         if (reverse) {
129             on = -1;
130         }
131
132         // 模拟合并过程，一开始，从长度为一合并到长度为二，一直合并到长
                度为
133         // len。
134         for (size_t h = 2; h <= len; h <<= 1) {
135             // wn：当前单位复根的间隔：w^1_h
136             Complex wn(cos(2 * PI / (double)h), sin(on * 2 * PI / (
                    double)h));
137             // 合并，共 len / h 次。
138             for (size_t j = 0; j < len; j += h) {
139                 // 计算当前单位复根，一开始是 1 = w^0_n，之后是以 wn
140                 // 为间隔递增： w^1_n
141                 // ...
142                 Complex w(1, 0);
143                 for (size_t k = j; k < j + h / 2; k++) {
144                     // 左侧部分和右侧是子问题的解
145                     Complex u = y[k];
146                     Complex t = w * y[k + h / 2];
147                     // 这就是把两部分分治的结果加起来
148                     y[k] = u + t;
149                     y[k + h / 2] = u - t;
150                     // 后半个「step」中的ω一定和「前半个」中的成相
                            反数
151                     // 「红圈」上的点转一整圈「转回来」，转半圈正好转成
                            相反数
152                     // 一个数相反数的平方与这个数自身的平方相等
153                     w = w * wn;
154                 }
155             }
156         }
157         // 如果是 IDFT，它的逆矩阵的每一个元素不只是原元素取倒数，还要
                除以长度
158         // len。
159         if (on == -1) {
160             for (size_t i = 0; i < len; i++) {
161                 y[i].real(y[i].real() / (double)len);
162             }
163         }
164     }
165 };
166
167 int main(void) {
168     std::vector<number> a_digit_list = get_big_int();
```

```
169     std::vector<number> b_digit_list = get_big_int();
170
171     Multiplier mul{a_digit_list, b_digit_list};
172
173     std::vector<number> temp = mul.multiply();
174
175     std::vector<int> result;
176     result.resize(2 * temp.size(), 0);
177
178     int carry = 0;
179
180     for (size_t i = 0; i < temp.size(); i++) {
181         int current_digit = (int)(temp[i] + 0.5);
182
183         current_digit += carry;
184
185         result[i] = current_digit % 10;
186         carry = current_digit / 10;
187     }
188
189     size_t result_len = 0;
190
191     for (size_t i = temp.size(); i < result.size(); i++) {
192         int current_digit = carry;
193
194         result[i] = current_digit % 10;
195         carry = current_digit / 10;
196
197         if (carry == 0 && result[i] == 0) {
198             result_len = i;
199             break;
200         }
201     }
202
203     for (size_t i = 0; i < result_len; i++) {
204         std::putchar(result[result_len - i - 1] + '0');
205     }
206
207     std::putchar('\n');
208
209     return 0;
210 }
211
212 std::vector<number> get_big_int() {
213     std::vector<number> result;
214     std::scanf("%s", buffer);
215
216     size_t num_len = std::strlen(buffer);
217
218     result.resize(num_len, 0);
219
```

```
    for (size_t i = 0; i < num_len; i++) {
        result[num_len - i - 1] = (number)(buffer[i] - '0');
    }

    return result;
}
```

数论

高精度

```
#include <algorithm>
#include <cctype>
#include <cstdio>
#include <string>
#include <vector>

struct BigInt {
    std::vector<int> digits;
    size_t len;

    explicit BigInt(size_t capacity) {
        digits.resize(capacity, 0);
        len = 1;
    };

    explicit BigInt(size_t capacity, long long from) {
        digits.resize(capacity, 0);

        size_t i = 0;

        while (from > 0) {
            digits[i] = (int)(from % 10);
            from /= 10;
            i++;
        }

        len = i;

        if (len == 0) {
            len = 1;
        }
    }

    explicit BigInt(size_t capacity, std::string const &from) {
        size_t valid_len = from.size();

        for (size_t i = 0; i < from.size(); i++) {
```

```
            if (!std::isalnum(from[i])) {
                valid_len = i;
                break;
            }
        }

        digits.resize(capacity, 0);

        for (size_t i = 0; i < valid_len; i++) {
            digits[i] = from[from.size() - i - 1] - '0';
        }

        len = valid_len;

        if (len == 0) {
            len = 1;
        }
    }

    friend BigInt operator+(BigInt const &left, BigInt const &right) {
        BigInt result{std::max(left.digits.size(), right.digits.size())
            };

        size_t new_len = 0;

        if (left.len > right.len) {
            std::copy_n(std::begin(left.digits), left.len,
                        std::begin(result.digits));
            for (size_t i = 0; i < right.len; i++) {
                result.digits[i] += right.digits[i];
            }

            new_len = left.len;
        } else {
            std::copy_n(std::begin(right.digits), right.len,
                        std::begin(result.digits));

            for (size_t i = 0; i < left.len; i++) {
                result.digits[i] += left.digits[i];
            }

            new_len = right.len;
        }

        int carry = 0;
        for (size_t i = 0; i < result.digits.size(); i++) {
            result.digits[i] += carry;
            carry = result.digits[i] / 10;
            result.digits[i] %= 10;

            if (i >= new_len && carry == 0 && result.digits[i] == 0) {
```

```
                new_len = i;
                break;
            }
        }

        result.len = new_len;

        if (result.len == 0) {
            result.len = 1;
        }
        return result;
    }

    friend BigInt operator-(BigInt const &left, BigInt const &right) {
        BigInt result{left.digits.capacity()};

        std::copy_n(std::begin(left.digits), left.len,
                    std::begin(result.digits));

        for (size_t i = 0; i < left.len; i++) {
            result.digits[i] -= right.digits[i];
        }

        int borrow = 0;

        for (size_t i = 0; i < left.len; i++) {
            result.digits[i] -= borrow;

            if (result.digits[i] < 0) {
                borrow = -result.digits[i] / 10 + 1;
                result.digits[i] += borrow * 10;
            }
        }

        for (size_t i = 0; i < left.len; i++) {
            size_t idx = (left.len - i - 1);

            if (result.digits[i] != 0) {
                result.len = idx + 1;
                break;
            }
        }

        if (result.len == 0) {
            result.len = 1;
        }

        return result;
    }

    friend BigInt operator*(BigInt const &left, BigInt const &right) {
```

```
139         BigInt result{std::max(left.digits.size(), right.digits.size())
                };
140 
141         for (size_t i = 0; i < left.len; i++) {
142             for (size_t j = 0; j < right.len; j++) {
143                 result.digits[i + j] += left.digits[i] * right.digits[j
                        ];
144             }
145         }
146 
147         int carry = 0;
148         for (size_t i = 0; i < result.digits.size(); i++) {
149             result.digits[i] += carry;
150             carry = result.digits[i] / 10;
151             result.digits[i] %= 10;
152         }
153 
154         result.len = 1;
155 
156         for (size_t i = 0; i < (left.len + right.len + 1); i++) {
157             size_t idx = left.len + right.len - i;
158 
159             if (result.digits[idx] != 0) {
160                 result.len = idx + 1;
161                 break;
162             }
163         }
164 
165         return result;
166     }
167 
168     std::string to_string() {
169         std::string s;
170 
171         s.resize(len, '0');
172 
173         for (size_t i = 0; i < len; i++) {
174             s[len - i - 1] = (char)digits[i] + '0';
175         }
176 
177         return s;
178     }
179 };
180 
181 int main(void) {
182     long long a, b;
183     a = b = 0;
184 
185     std::scanf("%lld%lld", &a, &b);
186 
187     BigInt b_a{1000, a};
```

```
    BigInt b_b{1000, b};

    std::printf("a + b: %s\n", (b_a + b_b).to_string().data());
    std::printf("a - b: %s\n", (b_a - b_b).to_string().data());
    std::printf("a * b: %s\n", (b_a * b_b).to_string().data());

    return 0;
}
```

GCD

```
#include <cstdlib>
#include <tuple>
#include <utility>

typedef int number;

number gcd(number a, number b) {
    a = std::abs(a);
    b = std::abs(b);
    if (a < b) {
        std::swap(a, b);
    }

    while (b > 0) {
        number temp = b;
        b = a % b;
        a = temp;
    }

    return a;
}

// d = gcd(a, b) = ax + by
struct EEResult {
    number d;
    number x;
    number y;
};

EEResult exgcd(number a, number b) {
    a = std::abs(a);
    b = std::abs(b);

    number x = 1, y = 0;

    number x1 = 0, y1 = 1, a1 = a, b1 = b;
    while (b1 > 0) {
        number q = a1 / b1;
```

```
        std::tie(x, x1) = std::make_tuple(x1, x - q * x1);
        std::tie(y, y1) = std::make_tuple(y1, y - q * y1);
        std::tie(a1, b1) = std::make_tuple(b1, a1 - q * b1);
    }
    return EEResult{a1, x, y};
}
```

线性同余方程

```
#include <cstdlib>
#include <tuple>

typedef int number;

// d = gcd(a, b) = ax + by
struct EEResult {
    number d;
    number x;
    number y;
};

EEResult exgcd(number a, number b) {
    a = std::abs(a);
    b = std::abs(b);

    number x = 1, y = 0;

    number x1 = 0, y1 = 1, a1 = a, b1 = b;
    while (b1 > 0) {
        number q = a1 / b1;
        std::tie(x, x1) = std::make_tuple(x1, x - q * x1);
        std::tie(y, y1) = std::make_tuple(y1, y - q * y1);
        std::tie(a1, b1) = std::make_tuple(b1, a1 - q * b1);
    }
    return EEResult{a1, x, y};
}

// Solve ax = b (mod n) for x_0
// x_i = x_0 + i * (n / d), d = gcd(a, n)
// solvable if and only if: d | b

struct MLEResult {
    bool solvable;
    number x0;
};

MLEResult modular_linear_equation_solver(number a, number b, number n)
    {
    EEResult r = exgcd(a, n);
```

```

    number d = r.d;

    if (b % d == 0) {
        return MLEResult{true, ((r.x * b / d) + n) % n};
    } else {
        return MLEResult{false, 0};
    }
}
```

模下快速幂

```
typedef long long number;

// a ^ b mod n
number modular_exponentitaion(number a, number b, number n) {
    number result = 1;
    number temp = a % n;

    while (b > 0) {
        number current_digit = b % 2;

        if (current_digit == 1) {
            result = (result * temp) % n;
        }

        temp = (temp * temp) % n;

        b /= 2;
    }

    return result;
}
```

CRT

```
#include <cstdlib>
#include <functional>
#include <numeric>
#include <tuple>
#include <vector>

typedef long long number;

// d = gcd(a, b) = ax + by
struct EEResult {
    number d;
```

```
12      number x;
13      number y;
14  };
15
16  EEResult exgcd(number a, number b) {
17      number x = 1, y = 0;
18
19      number x1 = 0, y1 = 1, a1 = a, b1 = b;
20      while (b1 > 0) {
21          number q = a1 / b1;
22          std::tie(x, x1) = std::make_tuple(x1, x - q * x1);
23          std::tie(y, y1) = std::make_tuple(y1, y - q * y1);
24          std::tie(a1, b1) = std::make_tuple(b1, a1 - q * b1);
25      }
26      return EEResult{a1, x, y};
27  }
28
29  // Solve ax = b (mod n) for x_0
30  // x_i = x_0 + i * (n / d), d = gcd(a, n)
31  // solvable if and only if: d | b
32
33  struct MLEResult {
34      bool solvable;
35      number x0;
36  };
37
38  MLEResult modular_linear_equation_solver(number a, number b, number n)
        {
39      EEResult r = exgcd(a, n);
40
41      number d = r.d;
42
43      if (b % d == 0) {
44          return MLEResult{true, ((r.x * b / d) + n) % n};
45      } else {
46          return MLEResult{false, 0};
47      }
48  }
49
50  // n = n_1n_2...n_k
51  // a = (a_1c_1 + a_2c_2 + ... + a_nc_n) mod n
52  struct CRTResult {
53      number a;
54      number n;
55  };
56
57  // n_1, n_2, ..., n_k are pairwise relative prime
58  // 求同余方程组：x = a_i (mod n_i)的解：x = a (mod n)
59  CRTResult crt(std::vector<number> const &a_list,
60                std::vector<number> const &n_list) {
61      number n = std::accumulate(std::begin(n_list), std::end(n_list), 1
```

```
        LL,
                                   std::multiplies<number>{});

    number result = 0;
    for (size_t i = 0; i < n_list.size(); i++) {
        number n_i = n_list[i];
        number m_i = n / n_i;

        MLEResult r = modular_linear_equation_solver(m_i, 1, n_i);

        number c_i = (r.x0 % n * m_i % n) % n;

        result = (result + (a_list[i] * c_i) % n) % n;
    }

    return CRTResult{result, n};
}
```

字符串匹配

KMP

```
#include <string>
#include <vector>

#define MAX_STR_LEN 305
#define ALPHABET_LEN 30

// 状态q表示匹配了pattern的前q个字符
class KMPMatcher {
  public:
    std::string pattern;
    std::vector<size_t> pi;

    explicit KMPMatcher(std::string const &pattern) {
        this->pattern = pattern;

        generate_pi();
    }

    void generate_pi() {
        pi.resize(pattern.size() + 1, 0);

        size_t q = 0;

        for (size_t i = 1; i < pattern.size(); i++) {
            while (q > 0 && pattern[i] != pattern[q]) {
                q = pi[q];
```

```
            }

            if (pattern[i] == pattern[q]) {
                q++;
            }

            pi[i + 1] = q;
        }
    }

    // O(N \sigma)
    // verdict: E6 D
    void generate_FA_transition_table(size_t delta[MAX_STR_LEN][
       ALPHABET_LEN]) {
        for (size_t l = 0; l < pi.size(); l++) {
            for (size_t next_ch = 'a'; next_ch <= 'z'; next_ch++) {
                size_t q = l;
                while ((q > 0) && ((char)next_ch != pattern[q])) {
                    q = pi[q];
                }

                if ((char)next_ch == pattern[q]) {
                    delta[l][next_ch - 'a'] = q + 1;
                } else {
                    delta[l][next_ch - 'a'] = q;
                }
            }
        }
    }

    std::vector<size_t> match(std::string const &to_match) {
        std::vector<size_t> matched_pos_list;

        size_t q = 0;

        for (size_t i = 0; i < to_match.size(); i++) {
            while (q > 0 && to_match[i] != pattern[q]) {
                q = pi[q];
            }

            // 0 下标，故已经匹配了q个字符的情况下，下一个比较的字符是
               pattern[q]
            if (to_match[i] == pattern[q]) {
                q++;
            }

            if (q == pattern.size()) {
                matched_pos_list.push_back(i + 1 - pattern.size());
                q = pi[q];
            }
        }
```

```

        return matched_pos_list;
    }
};

int main(void) {
    std::string pattern{"ababaca"};

    KMPMatcher matcher{pattern};

    return 0;
}
```

计算几何

```cpp
#include <algorithm>
#include <cmath>
#include <vector>

#define MY_INFINITY (1LL << 60)

typedef long long number;

class Point {
  public:
    number x;
    number y;

    explicit constexpr Point(number a, number b) : x{a}, y{b} {};

    friend Point operator-(Point const &left, Point const &right) {
        return Point{left.x - right.x, left.y - right.y};
    }

    friend bool operator==(Point const &left, Point const &right) {
        return (left.x == right.x) && (left.y == right.y);
    }
};

struct PointHash {
    std::size_t operator()(Point const &k) const noexcept {
        size_t right = (size_t)k.y;

        right += 0x9e3779b97f4a7c15;

        right = ((right << 31) | (right >> (64 - 31)));

        return ((size_t)k.x ^ right);
    }
```

```
};

class Vector {
  public:
    number x;
    number y;

    explicit constexpr Vector(number a, number b) : x{a}, y{b} {};
    explicit constexpr Vector(Point const &point) : x{point.x}, y{point
        .y} {};
    explicit constexpr Vector(Point const &from, Point const &to)
        : x{to.x - from.x}, y{to.y - from.y} {};

    friend Vector operator+(Vector const &left, Vector const &right) {
        return Vector{left.x + right.x, left.y + right.y};
    }

    friend Vector operator-(Vector const &left, Vector const &right) {
        return Vector{left.x - right.x, left.y - right.y};
    }

    friend Vector operator-(Vector const &self) {
        return Vector{-self.x, -self.y};
    }

    friend number operator*(Vector const &left, Vector const &right) {
        return left.x * right.x + left.y * right.y;
    }

    friend number operator^(Vector const &left, Vector const &right) {
        return left.x * right.y - right.x * left.y;
    }

    double module() const {
        return std::sqrt(this->x * this->x + this->y * this->y);
    }
};

class Segment {
  public:
    Point from;
    Point to;

    explicit constexpr Segment(Point a, Point b) : from{a}, to{b} {};

    Vector direction() const { return Vector{to.x - from.x, to.y - from
        .y}; }

    Segment reversed() const { return Segment{to, from}; }

    bool on_segment(Point p) const {
```

```
84          Vector d = this->direction();
85
86          Vector d2 = Vector{p - this->from};
87
88          if ((d ^ d2) == 0) {
89              if (p.x >= std::min(from.x, to.x) &&
90                  p.x <= std::max(from.x, to.x) &&
91                  p.y >= std::min(from.y, to.y) &&
92                  p.y <= std::max(from.y, to.y)) {
93                  return true;
94              }
95          }
96
97          return false;
98      }
99
100     bool intersect(Segment const &s) const {
101         Vector this_direction = this->direction();
102         Vector s_direction = s.direction();
103
104         Vector d1 = Vector{s.from - this->from};
105         Vector d2 = Vector{s.from - this->to};
106         Vector d3 = Vector{this->from - s.from};
107         Vector d4 = Vector{this->from - s.to};
108
109         number pro1 = s_direction ^ d1;
110         number pro2 = s_direction ^ d2;
111         number pro3 = this_direction ^ d3;
112         number pro4 = this_direction ^ d4;
113
114         if (((pro1 > 0 && pro2 < 0) || (pro1 < 0 && pro2 > 0)) &&
115             ((pro3 > 0 && pro4 < 0) || (pro3 < 0 && pro4 > 0))) {
116             return true;
117         }
118
119         if (pro1 == 0 && s.on_segment(this->from)) {
120             return true;
121         }
122
123         if (pro2 == 0 && s.on_segment(this->to)) {
124             return true;
125         }
126
127         if (pro3 == 0 && on_segment(s.from)) {
128             return true;
129         }
130
131         if (pro4 == 0 && on_segment(s.to)) {
132             return true;
133         }
134
```

```
135             return false;
136         }
137     };
138
139     // 按极角排序
140     bool operator<(Point const &left, Point const &right) {
141         number cross_product = Vector{left} ^ Vector { right };
142         if (cross_product > 0) {
143             return true;
144         } else if (cross_product == 0) {
145             if (left.x > right.x) {
146                 return false;
147             } else if (left.x < right.x) {
148                 return true;
149             } else {
150                 if (left.y > right.y) {
151                     return false;
152                 } else {
153                     return true;
154                 }
155             }
156         } else {
157             return false;
158         }
159     }
160
161     // 无需对point_list进行预处理
162     // 要求：不得含重复点！
163     std::vector<Point> convex_hull(std::vector<Point> point_list) {
164         number min_y = MY_INFINITY;
165         number min_x = MY_INFINITY;
166         auto origin_element = std::begin(point_list);
167
168         for (Point const &point : point_list) {
169             if (point.y < min_y) {
170                 min_y = point.y;
171             }
172         }
173
174         for (auto iter = std::begin(point_list); iter != std::end(
                point_list);
175              iter++) {
176             if (iter->y == min_y) {
177                 if (iter->x < min_x) {
178                     min_x = iter->x;
179                     origin_element = iter;
180                 }
181             }
182         }
183
184         for (Point &point : point_list) {
```

```
185         point.x -= min_x;
186         point.y -= min_y;
187     }
188 
189     point_list.erase(origin_element);
190 
191     std::sort(std::begin(point_list), std::end(point_list));
192     std::vector<Point> stack;
193 
194     stack.reserve(point_list.size());
195 
196     stack.emplace_back(0, 0);
197     stack.push_back(point_list[0]);
198     stack.push_back(point_list[1]);
199 
200     for (size_t i = 2; i < point_list.size(); i++) {
201         while (true) {
202             Point const &current_top = stack.back();
203             Point const &current_next_to_top = stack[stack.size() - 2];
204             Vector v1 = Vector{current_next_to_top, current_top};
205             Vector v2 = Vector{current_next_to_top, point_list[i]};
206 
207             if ((v1 ^ v2) > 0) {
208                 break;
209             }
210 
211             stack.pop_back();
212 
213             if (stack.size() <= 1) {
214                 break;
215             }
216         }
217 
218         stack.push_back(point_list[i]);
219     }
220 
221     return stack;
222 }
223 
224 // 求周长（若为线段，则为2*线段长度）
225 double length(std::vector<Point> const &convex_hull_list) {
226     double result = 0.0;
227 
228     size_t len = convex_hull_list.size();
229     for (size_t i = 0; i < len; i++) {
230         Point const &p1 = convex_hull_list[i];
231         Point const &p2 = convex_hull_list[(i + 1) % len];
232 
233         Vector v{p1, p2};
234 
235         result += v.module();
```

```
    }

    return result;
}

// 求多边形面积（返回2 * 面积）
// 注意转换为double除以2可能有精度问题！
number doubled_polygon_area(std::vector<Point> const &
    sorted_convex_list) {
    Point const &aux_ponint = *std::begin(sorted_convex_list);

    number doubled_area = 0;

    size_t len = sorted_convex_list.size();

    for (size_t i = 0; i < len; i++) {
        Vector v1{aux_ponint, sorted_convex_list[i]};
        Vector v2{aux_ponint, sorted_convex_list[(i + 1) % len]};

        doubled_area += (v1 ^ v2);
    }

    return std::abs(doubled_area);
}
```

Random

shuffle

```
#include <algorithm>
#include <random>
// 32 bit mersenne twister engine
auto const seed = std::random_device{}();
auto reng = std::mt19937{seed};
std::vector<int> v {0,1,2,3,4,5,6,7,8};
shuffle(begin(v)+2, begin(v)+7, reng);
for (int x : v) { cout << x <<' '; }  // 0 1 … 7 8
```

均匀分布

```
#include <random>
// fixed seed
auto const seed = 123;
// Mersenne Twister random engine:
std::mt19937 urbg {seed};
// generate random ints ∈ [1,6]
```

```
7  std::uniform_int_distribution<int> distr1 {1, 6};
8  auto const value1 = distr1(urbg);
9  auto const value2 = distr1(urbg);
10 // generate random floats ∈ [-1.2,6.25)
11 std::uniform_real_distribution<float> distr2 {-1.2f, 6.25f};
12 auto const value3 = distr2(urbg);
```