Ícaro Nunes - ICPC Library

Lembretes:

- Ler com cuidado
- Observar se arrays/segs/etc estão com tamanho de acordo com as constraints dadas e calculadas
- Checar se um grafo de entrada pode ter arestas repetidas
- Revisar assumptions feitas
- Observar casos base/edge

Formulas:

- Expected value of geometric distribution (expected number of failed Bernoulli trials before the first success, where p is the probability of a single success) $= \frac{1-p}{p}$
- Sum of arithemtic sequence:
  - $n \times \frac{(a_1+a_n)}{2}$
  - $\frac{n}{2} \times [2a_1 + (n-1) \times d]$
  - Where: $a_1 =$ first term, $d =$ common difference, $n =$ number of terms in the range
- Law of sines:
  - $\frac{sin(A)}{a} = \frac{sin(B)}{b} = \frac{sin(C)}{c}$
  - Where $x =$ length of side opposed to angle X
- Law of cosines:
  - $a^2 = b^2 + c^2 - 2bc \times cos(A)$
  - Where $a =$ length of side opposed to angle A

Contents

1 Data Structures

1.1 Mo's Algorithm

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

const int BLKSIZE=1e2;


// OPTIONAL
inline int64_t hilbertOrder(int x, int y, int pow, int rotate) {
	if (pow == 0) {
		return 0;
	}
	int hpow = 1 << (pow-1);
	int seg = (x < hpow) ? (
		(y < hpow) ? 0 : 3
	) : (
		(y < hpow) ? 1 : 2
	);
	seg = (seg + rotate) & 3;
	const int rotateDelta[4] = {3, 0, 0, 1};
	int nx = x & (x ^ hpow), ny = y & (y ^ hpow);
	int nrot = (rotate + rotateDelta[seg]) & 3;
	int64_t subSquareSize = int64_t(1) << (2*pow - 2);
	int64_t ans = seg * subSquareSize;
	int64_t add = hilbertOrder(nx, ny, pow-1, nrot);
	ans += (seg == 1 || seg == 2) ? add : (subSquareSize - add - 1);
	return ans;
}
//-------------

struct Query {
	int l, r, idx;
	int64_t ord;

	Query(int l, int r, int idx) : l(l), r(r), idx(idx) {
		ord = hilbertOrder(l, r, 21, 0);
	}

	bool operator < (const Query& other) const {
		// return ord < other.ord;
		int i1 = l/BLKSIZE, i2 = other.l/BLKSIZE;
		if(i1 == i2) return (i1%2 ? r > other.r : r < other.r);
		return i1 < i2;
	}
};

int add(int i) {
	// TODO
}


int remove(int i) {
	// TODO
}


void process(const vector<Query>& queries) {
	vector<int> res(queries.size());

	for(const auto& query: queries) {
		while(l > query.l)
			add(--l);

		while(r < query.r)
```

```
			add(++r);

		while(l < query.l)
			remove(l++);

		while(r > query.r)
			remove(r--);

		res[query.idx] = ans;
	}
}
```

1.2 Iterative Segment Tree

```
#include <vector>
#include <optional>

// segment tree - 0 indexed !
// space: 4n
// time:
// - range query: O(logn)
// - lower_bound: O(logn)
// - update: O(logn)
template<typename T, class Op>
struct STree {
	int size;
	int treesize;
	T* items;
	T* nodes;
	Op op;

	STree(const std::vector<T>& from) {
		treesize=1;

		while(treesize < from.size())
			treesize <<=1;

		this->size = from.size();
		nodes = new T[2*treesize];
		items = nodes+treesize;

		for(int i=0; i<2*treesize; i++)
			nodes[i] = op.neutral;

		for(int i=0; i<size; i++)
			items[i] = from[i];

		for(int i=2*treesize-1; i>1; i--)
			nodes[i>>1] = op(nodes[i>>1], nodes[i]);
	}

	void update(int index, T value) {
		int i = treesize+index;

		nodes[i] = value;
		i = i>>1;

		while(i>0) {
			nodes[i] = op(nodes[2*i], nodes[2*i+1]);
			i>>=1;
		}
	}

	T query(int l, int r) {
		T res = op.neutral;
		r++;

		for(l += treesize, r += treesize; l<r; l>>=1, r>>=1) {
			if(l&1) res = op(res, nodes[l++]);
			if(r&1) res = op(res, nodes[--r]);
		}

		return res;
	}

	~STree() {
		delete nodes;
	}

	// search for first ocurrence to the right of strt
	// of a value greater than or equal to v
	// REQUIRES TESTING ON STRT != 0
	int lower_boundr(int strt, T v) {
		int i=treesize+strt;
		T accum=nodes[i];
		while(i>1 and !(accum >= v)) {
			if(!(i&1) and op(accum, nodes[i+1])>=v) break;
			if(!(i&1)) accum = op(accum, nodes[i+1]);
			i>>=1;
		}

		if(i==1) return size;
		if(!(accum >= v)) i++;

		while(i<treesize) {
			if(op(accum, nodes[i<<1])>=v) i <<= 1;
			else accum = op(accum, nodes[i<<1]), i = (i<<1) + 1;
		}

		return i-treesize;
	}
};
```

1.3 Union-Find

```
// union find code
int parent[1123456];
int ranks[1123456];

int disjoint = 0;

// the use of ct and newSet()
// is useful in problems
// where the initial number
// of disjoint sets is unknown
// such as in Hoshen-Kopelman
int ct = 0;
int newSet() {
  ct++;
  disjoint++;
  parent[ct] = ct;
  ranks[ct] = 1;

  return ct;
}

// for "regular" disjoint-sets problems
// one must use this function to
// initialize the union_find
inline void init_union_find(int sz) {
  disjoint = sz;

  for(int i=1; i<=sz; i++) {
    parent[i] = i;
    ranks[i] = 1;
  }
}

// find using compression
int find(const int& v) {
  int root = parent[v];

  while(root != parent[root]) {
    parent[root] = parent[parent[root]];
    root = parent[root];
  }

  parent[v] = root;

  return root;
}

// merge using ranks
void merge(const int& a, const int& b) {
  int root_a = find(a);
  int root_b = find(b);

  if(root_a == root_b)
    return;

  if(ranks[root_b] > ranks[root_a]) {
    parent[root_a] = root_b;
    ranks[root_b] += ranks[root_a];
  } else {
    parent[root_b] = root_a;
    ranks[root_a] += ranks[root_b];
  }

  disjoint--;
}
// end union find code
```

2 Graph

2.1 Dijkstra

```
#include <algorithm>
#include <limits>
#include <queue>
#include <stack>
#include <vector>


template<typename T>
void dijkstra(const std::vector<std::pair<T, int>>* graph, bool* visited, T* dist, int* parent, int root
      , int n) {
  for(int i=1; i<=n; i++)
    dist[i] = std::numeric_limits<T>::max();

  std::priority_queue<std::pair<T, int>, std::vector<std::pair<T, int>>, std::greater<std::pair<T, int
      >>> pq;

  pq.push({ 0, root });
  dist[root] = 0;
  parent[root] = root;

  while(!pq.empty()) {
    auto [dst, node] = pq.top();
    pq.pop();

    if(visited[node])
      continue;

    visited[node] = true;

    const std::vector<std::pair<T, int>>& adj = graph[node];

    for(const auto& [c_dst, child]: adj) {
      if(visited[child])
        continue;

      T newdst = dist[node] + c_dst;

      if(dist[child] > newdst) {
        dist[child] = newdst;
        pq.push({ newdst, child });
      }
    }
  }
}


// retrieve_path SEEMS TO BE NOT WORKING
//
// std::vector<int> retrieve_path(int tgt, int* parent) {
//   std::vector<int> ans;
//   int curr = tgt;
//   ans.push_back(tgt);
//
//   while(curr != parent[curr]) {
//     ans.push_back(parent[curr]);
//     curr = parent[curr];
//   }
//
//   std::reverse(ans.begin(), ans.end());
//
//   return ans;
// }
```

2.2 Toposort

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

bool dfs(const std::vector<int>* graph, bool* visited, int* currtrav, std::stack<int>& ans, int node) {
        visited[node] = true;

        const std::vector<int>& adj = graph[node];

        bool poss = true;

        for(const int& child: adj) {
                if(currtrav[child] == currtrav[node]) {
                        poss = false;
                        break;
                }

                if(!visited[child]) {
                        currtrav[child] = currtrav[node];

                        if(!dfs(graph, visited, currtrav, ans, child)) {
                                poss = false;
                                break;
                        }
                }
        }

        if(!poss)
                return false;

        currtrav[node] = -1;
        ans.push(node);
        return true;
}

// dfs-based toposort
// returns an empty vector in case it finds a cycle
// (which means there is no valid topological order
// for the graph)
//
// PS. the array currtrav representes the origin node
// for the current traversal,it will be useful for
// finding if there are cycles
//
// PS2. removed std::optional since it only compiles on
// C++17 above, and judges such as SPOJ only supports up
// to C++14
std::vector<int> tarjan(const std::vector<int>* graph, bool* visited, int* currtrav, int n) {
        for(int i=1; i<=n; i++)
                currtrav[i] = -1;

        bool poss = true;

  std::stack<int> ans;

        for(int i=1; i<=n; i++) {
                if(visited[i])
                        continue;

                currtrav[i] = i;

                if(!dfs(graph, visited, currtrav, ans, i)) {
                        poss = false;
                        break;
                }
        }

        if(poss) {
                std::vector<int> v;

                while(!ans.empty()) {
                        v.push_back(ans.top());
                        ans.pop();
                }

                return v;
        } else
                return std::vector<int>();

}



// "bfs-based" toposort
// returns an empty vector in case it finds a cycle
// (which means there is no valid topological order
// for the graph)
//
// PS. removed std::optional since it only compiles on
// C++17 above, and judges such as SPOJ only supports up
// to C++14
std::vector<int> kahn(const std::vector<int>* graph, bool* visited, int* indeg, int n) {
  std::queue<int> q;

  for(int i=1; i<=n; i++) {
    if(indeg[i] == 0)
      q.push(i);
  }

  std::vector<int> ans;
```

```
  while(!q.empty()) {
    int node = q.front();
    q.pop();

    if(visited[node])
      continue;

    visited[node] = true;
    ans.push_back(node);

    const std::vector<int>& adj = graph[node];

    for(const int& child: adj) {
      indeg[child]--;

      if(indeg[child] == 0)
        q.push(child);
    }
  }

  if(ans.size() == n)
    return ans;
  else
    return std::vector<int>();
}
```

2.3 Prim/Kruskal

```
#include <queue>
#include <vector>
#include "../data-structure/union_find/union_find.h"

template<typename T>
T prim(const std::vector<std::pair<T, int>>* graph, bool* visited, int root) {
  using pTi = std::pair<T, int>;
  std::priority_queue<pTi, std::vector<pTi>, std::greater<pTi>> pq;

  T total = 0;

  pq.push({0, root});

  while(!pq.empty()) {
    auto [cost, node] = pq.top();
    pq.pop();

    if(visited[node])
      continue;

    visited[node] = true;
    total += cost;

    const std::vector<pTi>& adj = graph[node];

    for(const auto& [c_dist, child]: adj) {
      if(visited[child])
        continue;

      pq.push({c_dist, child});
    }
  }

  return total;
}


// for kruskal
template<typename T>
struct edge {
  int a, b;
  T c;

  bool operator<(const edge<T>& other) const {
    return c < other.c;
  }
};

template<typename T>
T kruskal(std::vector<edge<T>>& edges, int n) {

  init_union_find(n);

  sort(edges.begin(), edges.end());

  T total = 0;

  for(const edge<T>& e: edges) {
    int root_a = find(e.a);
    int root_b = find(e.b);

    if(root_a != root_b) {
      total += e.c;
      merge(root_a, root_b);
    }
  }

  return total;
}
```

2.4 Floyd-Warshall

```
#include <cstddef>
#include <limits>

template<typename T, std::size_t N, std::size_t M>
using Matrix = T[N][M];

// the adjacency matrix passed to floyd-warshall must contain
// weight of INF (numeric_limits<T>::max()) on pairs of nodes
// that are not adjacent, and 0 on the main diagonal
template<typename T, std::size_t N, std::size_t M>
void floydWarshall(const Matrix<T, N, M>& graph, Matrix<T, N, M>& dist, int n) {
  const T INF = std::numeric_limits<T>::max();

  for(int i=1; i<=n; i++)
    for(int j=1; j<=n; j++)
      dist[i][j] = graph[i][j];

  // k -> intermediate node
  for(int k=1; k<=n; k++) {
    // i -> source
    for(int i=1; i<=n; i++) {
      // j -> target
      for(int j=1; j<=n; j++) {
        T newdist;

        if(dist[i][k] == INF || dist[k][j] == INF)
          newdist = INF;
        else
          newdist = dist[i][k] + dist[k][j];

        dist[i][j] = min(dist[i][j], newdist);
      }
    }
  }
}
```

3 Math

3.1 FFT (recursive)

```
#include <iostream>
#include <math>

const long double PI = std::arccos(-1);

template<typename T>
struct Complex {
        T a, b;

        Complex(T a, T b) {
                this->a = a;
                this->b = b;
        }

        Complex(T a) {
                this->a = a;
                this->b = 0;
        }

        Complex operator+(const Complex& other) const {
                return Complex(a+other.a, b+other.b);
        }

        Complex operator-(const Complex& other) const {
                return Complex(a-other.a, b-other.b);
        }

        Complex operator*(const T& real) const {
```

```
		return Complex(a*real, b*real);
	}

	Complex operator*(const Complex& other) const {
		return Complex(a*other.a - b*other.b, a*other.b + b*other.a);
	}
};


// works for a power of 2 sized vector
// T must be a floating point type because
// of the trigonometric operations
template<typename T>
vector<Complex<T>> fft(const vector<T>& p) {
	int n = p.size();

	vector<Complex<T>> ans(n);

	if(n == 1) {
		ans[0] = Complex<T>(p[0]);
		return ans;
	}

	T angle = (2*PI)/n;
	Complex<T> w = Complex<T>(std::cos(angle), std::sin(angle));

	vector<T> pe(n/2), po(n/2);
	for(int i=0; i<=n-2; i+=2)
		pe[i/2] = p[i], po[i/2] = p[i+1];

	vector<Complex<T>> ye = fft(pe), yo = fft(po);

	Complex<T> w_i = Complex<T>(T(1));

	for(int i=0; i<n/2; i++) {
		ans[i] = ye[i] + w_i*yo[i];
		ans[i+n/2] = ye[i] - w_i*yo[i];
		w_i = w_i*w;
	}

	return ans;
}



// works for a power of 2 sized vector
// T must be a floating point type because
// of the trigonometric operations
template<typename T>
vector<Complex<T>> ifft(const vector<Complex<T>>& p) {
	int n = p.size();

	vector<Complex<T>> ans(n);

	if(n == 1) {
		ans[0] = Complex<T>(p[0]);
		return ans;
	}

	T angle = (2*PI)/n;
	Complex<T> w = (1/double(n)) * Complex<T>(std::cos(-angle), std::sin(-angle));

	vector<T> pe(n/2), po(n/2);
	for(int i=0; i<=n-2; i+=2)
		pe[i/2] = p[i], po[i/2] = p[i+1];

	vector<Complex<T>> ye = ifft(pe), yo = ifft(po);

	Complex<T> w_i = Complex<T>(T(1));

	for(int i=0; i<n/2; i++) {
		ans[i] = ye[i] + w_i*yo[i];
		ans[i+n/2] = ye[i] - w_i*yo[i];
		w_i = w_i*w;
	}

	return ans;
}




std::ostream& operator<<(std::ostream& os, const Complex<T>& c) {
	os << "Complex(" << c.a << ", " << c.b << ")";
	return os;
}
```

3.2 FFT (in-place)

```
using ld = long double;
using cd = complex<ld>;
const ld PI = acos(-1);

int reverse(int num, int lg_n) {
    int res = 0;
    for (int i = 0; i < lg_n; i++) {
        if (num & (1 << i))
            res |= 1 << (lg_n - 1 - i);
    }
    return res;
}

void fft(vector<cd> & a, bool invert) {
    int n = a.size();
    int lg_n = 0;
    while ((1 << lg_n) < n)
        lg_n++;

    for (int i = 0; i < n; i++) {
        if (i < reverse(i, lg_n))
            swap(a[i], a[reverse(i, lg_n)]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        ld ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }

    if (invert) {
        for (cd & x : a)
            x /= n;
    }
}
```

3.3 Matrix Multiplication

```
#include <cstddef>


template<typename T, std::size_t N>
using Matrix = T[N][N];

template<typename T, std::size_t N>
void cp(const Matrix<T, N>& src, Matrix<T, N>& tgt) {
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      tgt[i][j] = src[i][j];
}

template<typename T, std::size_t N>
void matmul(const Matrix<T, N>& a, const Matrix<T, N>& b, Matrix<T, N>& dst, const int MOD) {
  Matrix<T, N> tmp;

  for(int i=0; i<N; i++) {
    for(int j=0; j<N; j++) {
      T sum = 0;

      for(int l=0; l<N; l++) {
        sum += (a[i][l]*b[l][j])%MOD;
        sum %= MOD;
      }

      tmp[i][j] = sum;
    }
  }

  cp(tmp, dst);
}
```

4 Number Theory

4.1 Divisors

```
#include <bits/stdc++.h>

// regular implementation
// - time complexity: O(sqrt(n))
// - space complexity: O(sqrt(n))
template<typename Integer>
std::vector<Integer> find_divisors(Integer val) {
  std::vector<Integer> v;
  std::stack<Integer> finals;

  Integer root = sqrt(val);

  // this less than or equal to is important
  // to avoid edge case like in find_divisors(20),
  // where sqrt(20) == 4, and therefore, in a
  // mistake I previously made, my algorithm was
  // checking "root" separately, only to check
  // if the square root was exact. Even though
  // 4 is not the exact square root of 20, it is
  // still a divisor of 20, and that version of
  // the algorithm would output { 1, 2, 10, 20 }
  // instead of { 1, 2, 4, 5, 10, 20 }. Another
  // fix was to keep the separate check, but
  // using ceil-rounded (instead of floor-rounded)
  // square root.
  for(Integer i=1; i<=root; i++) {
    if(val%i == 0) {
      v.push_back(i);

      Integer div = val/i;

      if(div != i)
        finals.push(div);
    }
  }

  while(!finals.empty()) {
    v.push_back(finals.top());
    finals.pop();
  }

  return v;
}
```

4.2 Fexp/Modular Inverse

```
template<typename T>
T fexp(T x, T exp, const T MOD) {
  T cursor = 1;
  T ans = 1;
  T currpow = x;

  while(cursor <= exp) {
    if(cursor & exp)
      ans = (ans%MOD) * (currpow%MOD) %MOD;

    cursor <<= 1;
    currpow = currpow%MOD;
    currpow = (currpow*currpow)%MOD;
  }

  return ans;
}

template<typename T>
T mod_inv(T x, const T MOD) {
  return fexp(x, MOD-1, MOD);
}
```

4.3 Gcd/Lcm

```
template<typename T>
T gcd(T a, T b) {
  if(a == b)
    return a;

  if(b > a)
    return gcd(b, a);

  if(!b) return a;
  return gcd(b, a%b);
}

template<typename T>
T lcm(T a, T b) {
  return a*b/gcd(a, b);
}
```

4.4 Sieve/Factors

```
#include <cmath>
#include <vector>

void sieve(bool* prime, std::vector<int>& primes, const int MAX_V) {
        for(int i=2; i<MAX_V; i++) {
                prime[i] = true;
        }

        int root = std::sqrt(MAX_V);

        for(int i=2; i<=root; i++) {
                if(!prime[i])
                        continue;

                primes.push_back(i);

                for(int j=i; j <= MAX_V; j+=i) {
                        prime[j] = false;
                }
        }

        for(int i=root; i<MAX_V; i++) {
                if(prime[i])
                        primes.push_back(i);
        }
}

std::vector<std::pair<int, int>> factors(const int& x, const std::vector<int>& primes) {
  std::vector<std::pair<int, int>> res;

        int rem = x;

        for(const int& p: primes) {
                if(p > rem)
                        break;

                if(rem % p != 0)
                        continue;

                int amt = 0;

                while(rem % p == 0) {
                        rem /= p;
                        amt++;
                }

                res.push_back(std::make_pair(p, amt));
        }

        return res;
}
```

5 Others

5.1 Kadane

```
template<typename T>
T kadane(T* arr, int n) {
  T curr = arr[0];
  T best = curr;

  for(int i=1; i<n; i++) {
    curr = max(arr[i], curr + arr[i]);
    best = max(best, curr);
  }

  return best;
}
```

5.2 Z Function

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

std::vector<int> zstring(const std::string& s) {
	std::vector<int> z(s.size(), 0);

	int left = 0, right = 0;

	for(int k=1; k < s.size(); k++) {
		if(k > right) {
			left = right = k;

			while(right < s.size() && s[right] == s[right - left])
				right++;

			z[k] = right - left;

			right--;
		} else /* k <= right */ {
			int k1 = k - left;

			if(z[k1] < right - k) {
				z[k] = z[k1];
			} else {
				left = k;

				while(right < s.size() && s[right] == s[right - left])
					right++;

				z[k] = right - left;
				right--;
			}
		}
	}

	return z;
}

std::vector<int> match(const std::string& s, const std::string& pattern) {
	std::string pt = pattern + "$" + s;
	std::vector<int> z = zstring(pt);
	std::vector<int> res;

	for(int i=pattern.size() + 1; i < pt.size(); i++) {
		if(z[i] == pattern.size())
			res.push_back(i - pattern.size() - 1);
	}

	return res;
}
```

6 Utils

6.1 Tester

```python
import random
import subprocess

MAX_N = 100


def gen_case() -> str:
    return f"1\n"

random.seed((1 << 9) | 31)

for i in range(100):
    print()
    print()
    case = gen_case()
    print(f"Test #{i+1}: ")
    print(case)
    # test bruteforce
    bf = subprocess.run(['out/b'], input=case, encoding='ascii', capture_output=True)
    # test solution
    sol = subprocess.run(['out/m'], input=case, encoding='ascii', capture_output=True)

    bf_res = bf.stdout
    sol_res = sol.stdout

    print(f"bruteforce {bf_res}, solution {sol_res}")

    if bf_res == sol_res:
        print("accepted")
    else:
        print("WA")
        break
```