University of Copenhagen

3 little 3 late

Asbjørn Lind, Elias Rasmussen Lolck, Thor Vejen Eriksen

NWERC 2025

November 23, 2025

Setup

hash.sh

```
# hashes a file, ignoring whitespaces and comments
# use for verifying that code is copied correctly
cpp -dD -P -fpreprocessed | tr -d '[:space:]' | md5sum |
    cut -c-6
```

vimrc

```
se ch=1 ic mouse=a sw=4 ts=4 nu rnu nuw=4 nowrap so=6
    siso=8 fdm=indent fdl=99 tm=100
ca Hash w !cpp -dD -P -fpreprocessed \| tr -d '[:space
    :]' \| md5sum \| cut -c-6
vnoremap <silent> p "_dP
vnoremap <silent> <A-Down> :m '>+1<CR>gv=gv
vnoremap <silent> <A-Up> :m '<-2<CR>gv=gv
```

Combinatorial

Permutation to Int

Description: [kactl] Given a permutation, returns the number of lexicographically strictly smaller permutations.
Complexity: O(n), but returns a value that is O(n!)

```
int permToInt(vector<int> v) {
  int use = 0, i = 0, r = 0;
  for(int x : v) {
    r = r * ++i + __builtin_popcount(use & -(1<<x));
    use |= 1 << x;
  }
  return r;
}
```

Multinomial

Description: [kactl] Computes $\binom{k_1+\cdots+k_n}{k_1,k_2,\ldots,k_n} = \frac{(\sum k_i)!}{k_1!k_2!\ldots k_n!}$.

```
ll multinomial(vector<int> v) {
  ll c = 1, m = v.size() ? v[0] : 1;
  for (int i = 1; i < (int)v.size(); i++)
    for (int j = 0; j < v[i]; j++)
      c = c * ++m / (j+1);
  return c;
}
```

Data_structures

Fenwick tree

Description: Computes prefix sums and single element updates. Uses 0-indexing.
Usage: `Fen f(n); f.update(ind, val); f.query(ind); f.lower_bound(sum);`
Complexity: $\mathcal{O}(\log n)$ per update/query

```
struct Fen {
  vector <ll> v;
  Fen(int s) : v(s, 0) { }
  void update(int ind, ll val) {
    for (; ind < (int) v.size(); ind |= ind + 1) v[ind]
        += val;
  }
  ll query(int ind) { // [0, ind), ind < 0 returns 0
    ll res = 0;
    for (; ind > 0; ind &= ind - 1) res += v[ind - 1];
    // operation can be modified
    return res;
  }
  int lower_bound(ll sum) { // returns first i with
    query(i + 1) >= sum, n if not found
    int ind = 0;
    for (int p = 1 << 25; p; p >>= 1) // 1 << 25 can be
    lowered to ceil(log2(v.size()))
      if (ind + p <= (int) v.size() && v[ind + p - 1] <
    sum)
        sum -= v[(ind += p) - 1];
    return ind;
  }
};
```

Li-Chao tree

Description: Contianer of lines, online insertion/querying. Retrieve the line f with minimum $f(x)$ for a given x.
Usage: `LCT lct(n); lct.insert(line, 0, n - 1); lct.query(x, 0, n - 1);`
Complexity: $\mathcal{O}(\log n)$ per insertion/query

```
struct Line { ll a, b; ll f(ll x) { return a * x + b; }
    };
constexpr const Line LINF { 0, 1LL << 60 };
struct LCT {
  vector <Line> v; // coord-compression: modify v[x] ->
    v[conert(x)]
  LCT(int size) { v.resize(size, LINF); }
  void insert(Line line, int l, int r) {
    if (l > r) return;
    int mid = (l + r) >> 1;
    if (line.f(mid) < v[mid].f(mid)) swap(line, v[mid]);
    if (line.f(l) < v[mid].f(l)) insert(line, l, mid -
    1);
    else insert(line, mid + 1, r);
  }
  Line query(int x, int l, int r) {
    if (l > r) return LINF;
    int mid = (l + r) >> 1;
    if (x == mid) return v[mid]; // faster on avg. - not
    necessary
    if (x < mid) return best_of(v[mid], query(x, l, mid
    - 1), x);
    return best_of(v[mid], query(x, mid + 1, r), x);
  }
  Line best_of(Line a, Line b, ll x) { return a.f(x) < b
    .f(x) ? a : b; }
};
```

Fast hash map

Description: 3x faster hash map, 1.5x more memory usage, similar API to std::unordered_map. Initial capacity, if provided, must be power of 2.
Usage: `hash_map <key_t, val_t> mp; mp[key] = val; mp.find(key); mp.begin(); mp.end(); mp.erase(key); mp.size();`
Complexity: $\mathcal{O}(1)$ per operation on average.

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

struct chash {
  const uint64_t C = ll(4e18 * acos(0)) | 71;
  ll operator () (ll x) const { return __builtin_bswap64
      (x * C); }
};
template <typename KEY_T, typename VAL_T> using hash_map
    = __gnu_pbds::gp_hash_table <KEY_T, VAL_T, chash>;
```

Persistent segment tree

Description: Zero-indexed, bounds are [l, r), operations can be modified. `update(...)` returns a pointer to a new tree with the applied update, all other trees remain unchanged. $\mathcal{O}(\log n)$ find_first and the like can be implemented by checking bounds, then checking left tree, then right tree, recursively.
Usage: `Node* root = build(arr, 0, n); Node* another_root = update(root, ind, val, 0, n); query(some_root, l, r, 0, n).val; Node* empty_root = nullptr; Node* another_version = update(empty_root, ind, val, 0, n);`
Complexity: $\mathcal{O}(\log n)$ per update/query, $\mathcal{O}(n)$ per build

```
struct Node {
  Node* l,* r;
  int val; // i.e. data
  Node(int _v) : l(nullptr), r(nullptr), val(_v) { }
  Node(Node* _l, Node* _r) : l(_l), r(_r), val(0) {
    // i.e. merge two nodes:
    if (l) val += l->val;
    if (r) val += r->val;
  }
};

// slightly more memory, much faster:
template <typename... ARGS> Node* new_node(ARGS&&...
    args) {
  static deque <Node> pool;
  pool.emplace_back(forward <ARGS> (args)...);
  return &pool.back();
}
// slightly less memory, much slower:
// #define new_node(...) new Node(__VA_ARGS__)

// optional:
Node* build(const vector <int>& a, int l, int r) {
  if (!(r - l - 1)) return new_node(a[l]);
  int mid = (l + r) >> 1;
  return new_node(build(a, l, mid), build(a, mid, r));
}

// can be called with node == nullptr
Node* update(Node* node, int ind, int val, int l, int r)
    {
  if (!(r - l - 1)) return new_node(val); // i.e. point
    update
  int mid = (l + r) >> 1;
  Node* lf = node ? node->l : nullptr;
  Node* rg = node ? node->r : nullptr;
  return new_node
    (ind < mid ? update(lf, ind, val, l, mid) : lf,
    ind >= mid ? update(rg, ind, val, mid, r) : rg);
}
Node query(Node* node, int tl, int tr, int l, int r) {
  if (l >= tr || r <= tl || !node) return Node(0); // i.
    e. empty node
  if (l >= tl && r <= tr) return *node;
  int mid = (l + r) >> 1;
  Node lf = query(node->l, tl, tr, l, mid);
  Node rg = query(node->r, tl, tr, mid, r);
  return Node(&lf, &rg);
}
```

Wavelet tree

Description: Taken from `https://ideone.com/Tkters`. k-th smallest element in a range. Count number of elements less than or equal to k in a range. Count number of elements equal to k in a range.
Usage: `wavelet_tree wt(arr, arr+n, 1, 1000000000); wt.kth(l, r, k); wt.LTE(l, r, k); wt.count(l, r, k);`
Complexity: $\mathcal{O}(\log n)$ per query

364273

```
struct wavelet_tree{
  #define vi vector<int>
  #define pb push_back
  int lo, hi;
  wavelet_tree *l, *r;
  vi b;

  //nos are in range [x,y]
  //array indices are [from, to)
  wavelet_tree(int *from, int *to, int x, int y){
    lo = x, hi = y;
    if(lo == hi or from >= to) return;
    int mid = (lo+hi)/2;
    auto f = [mid](int x){
      return x <= mid;
    };
    b.reserve(to-from+1);
    b.pb(0);
    for(auto it = from; it != to; it++)
      b.pb(b.back() + f(*it));
    //see how lambda function is used here
    auto pivot = stable_partition(from, to, f);
    l = new wavelet_tree(from, pivot, lo, mid);
    r = new wavelet_tree(pivot, to, mid+1, hi);
  }

  //kth smallest element in [l, r]
  int kth(int l, int r, int k){
    if(l > r) return 0;
    if(lo == hi) return lo;
    int inLeft = b[r] - b[l-1];
    int lb = b[l-1]; //amt of nos in first (l-1) nos that go in left
    int rb = b[r]; //amt of nos in first (r) nos that go in left
    if(k <= inLeft) return this->l->kth(lb+1, rb , k);
    return this->r->kth(l-lb, r-rb, k-inLeft);
  }

  //count of nos in [l, r] Less than or equal to k
  int LTE(int l, int r, int k) {
    if(l > r or k < lo) return 0;
    if(hi <= k) return r - l + 1;
    int lb = b[l-1], rb = b[r];
    return this->l->LTE(lb+1, rb, k) + this->r->LTE(l-lb, r-rb, k);
  }

  //count of nos in [l, r] equal to k
  int count(int l, int r, int k) {
    if(l > r or k < lo or k > hi) return 0;
    if(lo == hi) return r - l + 1;
    int lb = b[l-1], rb = b[r], mid = (lo+hi)/2;
    if(k <= mid) return this->l->count(lb+1, rb, k);
    return this->r->count(l-lb, r-rb, k);
  }
  ~wavelet_tree(){
    delete l;
    delete r;
  }
};
```

Strings

Rolling Hash

Description: RH prepare string s, and hash gives the hash of the substring [l, r] inclusive. ib is pow(b, -1, MD), MD should be prime
Complexity: $\mathcal{O}(n)$ preprocessing, $\mathcal{O}(1)$ hash.

2e25f9

```
struct RH {
  int MD, n, b, ib; // b is base, ib inverse base mod MD
  vector<int> p, ip, hs;
  RH(string s, int _b = 69, int _ib = 579710149, int _MD = 1e9 + 7) : MD(_MD), n((int)s.size()), b(_b), ib(_ib), p(n), ip(n), hs(n) { // _b = 63, _ib = 698412843, MD = 1e9 + 207
    p[0] = ip[0] = 1;
    hs[0] = s[0];
    for(int i = 1; i < n; ++i){
      p[i] = (ll) p[i - 1] * b % MD;
      ip[i] = (ll) ip[i - 1] * ib % MD;
      hs[i] = ((ll) s[i] * p[i] + hs[i - 1]) % MD; // s[i] can be changed to some hash function
    }
  }
  int hash(int l, int r){
    return (ll) (hs[r] - (l ? hs[l - 1] : 0) + MD) * ip[l] % MD;
  }
};
```

Suffix automaton

Description: Standard suffix automaton. Does what you'd expect.
Usage: See example main function below. This was thrown in last minute from a working cses solution.
Complexity: $\mathcal{O}(\log n)$ per update/query

3d234e

```
struct SA {
  struct State {
    int length;
    int link;
    int next[26];
    int cnt;
    bool is_clone;
    int first_pos;
    State(int _length, int _link) :
    length(_length),
    link(_link),
    cnt(0),
    is_clone(false),
    first_pos(-1)
    {
      memset(next, -1, sizeof(next));
    }
  };
  std::vector <State> states;
  int size;
  int last;
  bool did_init_count;
  int str_len;
  bool did_init_css;
  SA() :
  states(1, State(0, -1)),
  size(1),
  last(0),
  did_init_count(false),
  str_len(0),
  did_init_css(false)
  { }
  void push(char c) {
    str_len++;
    did_init_count = false;
    did_init_css = false;
    int cur = size;
    states.resize(++size, State(states[last].length + 1, -1));
    states[cur].first_pos = states[cur].length - 1;
    int p = last;
    while (p != -1 && states[p].next[c - 'a'] == -1) {
      states[p].next[c - 'a'] = cur;
      p = states[p].link;
    }
    if (p == -1) {
      states[cur].link = 0;
    } else {
      int q = states[p].next[c - 'a'];
      if (states[p].length + 1 == states[q].length) {
        states[cur].link = q;
      } else {
        int clone = size;
        states.resize(++size, State(states[p].length + 1, states[q].link));
        states[clone].is_clone = true;
        memcpy(states[clone].next, states[q].next, sizeof(State::next));
        states[clone].first_pos = states[q].first_pos;
        while (p != -1 && states[p].next[c - 'a'] == q) {
          states[p].next[c - 'a'] = clone;
          p = states[p].link;
        }
        states[q].link = states[cur].link = clone;
      }
    }
    last = cur;
  }
  bool exists(const std::string& pattern) {
    int node = 0;
    int index = 0;
    while (index < (int) pattern.length() && states[node].next[pattern[index] - 'a'] != -1) {
      node = states[node].next[pattern[index] - 'a'];
      index++;
    }
    return index == (int) pattern.size();
  }
  int count(const std::string& pattern) {
    if (!did_init_count) {
      did_init_count = true;
      for (int i = 1; i < size; i++) {
        states[i].cnt = !states[i].is_clone;
      }
      std::vector <std::vector <int>> of_length(str_len + 1);
      for (int i = 0; i < size; i++) {
        of_length[states[i].length].push_back(i);
      }
      for (int l = str_len; l >= 0; l--) {
        for (int node : of_length[l]) {
          if (states[node].link != -1) {
            states[states[node].link].cnt += states[node].cnt;
          }
        }
      }
    }
    int node = 0;
    int index = 0;
    while (index < (int) pattern.length() && states[node].next[pattern[index] - 'a'] != -1) {
      node = states[node].next[pattern[index] - 'a'];
      index++;
```

```
edf68d     }
72ab54     return index == (int) pattern.size() ? states[node].
        cnt : 0;
f7682f   }
f397ab   int first_occ(const std::string& pattern) {
53dacd     int node = 0;
6bbd47     int index = 0;
442e13     while (index < (int) pattern.length() && states[node
        ].next[pattern[index] - 'a'] != -1) {
652cc2       node = states[node].next[pattern[index] - 'a'];
8e968d       index++;
ef6d88     }
a59113     return index == (int) pattern.size() ? states[node].
        first_pos - (int) pattern.size() + 1 : -1;
a65c30   }
9afeb2   size_t count_substrings() {
a7f74b     static std::vector <size_t> dp;
9e504d     if (!did_init_css) {
9a3afa       did_init_css = true;
fce801       dp = std::vector <size_t> (size, 0);
75426a       auto dfs = [&] (auto&& self, int node) -> size_t {
673f0b         if (node == -1) {
0b0f06           return 0;
9fa531         }
99b459         if (dp[node]) {
ac9ba2           return dp[node];
519c50         }
983e54         dp[node] = 1;
1d020f         for (int i = 0; i < 26; i++) {
2e5625           dp[node] += self(self, states[node].next[i]);
515699         }
02606f         return dp[node];
b1fb1b       };
a3a17c       dfs(dfs, 0);
d8b4f0     }
8b5414     return dp[0] - 1;
e1c0a8   }
db005c };
db005c
db005c // usage example: Repeating Substring submission on cses
        .fi
2f5768 int main() {
109b3e   std::ios::sync_with_stdio(0); std::cin.tie(0);
c0bcd4   std::string s; std::cin >> s;
c9c93c   int n; std::cin >> n;
0c8f98   SA sa;
3b67c6   for (char c : s) {
5bd287     sa.push(c);
27d539   }
c64da9   sa.count("");
66d2ad   int len = -1;
bb09b1   int ind = -1;
af0b43   for (int i = 1; i < sa.size; i++) {
f4d141     if (sa.states[i].cnt > 1) {
eb5645       if (len < sa.states[i].length) {
961e2f         len = sa.states[i].length;
becb1e         ind = sa.states[i].first_pos - len + 1;
5af6dc       }
3b9795     }
0f2256   }
f0ebc0   if (len == -1) {
de5034     std::cout << "-1\n";
c8c5ae     return 0;
a99b6e   }
f38c31   for (int i = 0; i < len; i++) {
0d86ab     std::cout << s[i + ind];
42f1ff   }
228fb9   std::cout << "\n";
3d234e }
```

Various

Xor basis

Description: Basis of vectors in Z_2^d

61b70d

```
bf37aa struct XB {
6ea8b3   vector<int> basis;
ae23d0   void ins(int mask) {
6f1850     for(auto &y : basis) {
24dad5       if(y < mask) swap(y, mask);
af22b6       mask = min(mask, mask ^ y);
241cda     }
5fc70a     if(mask) basis.push_back(mask); // if mask is 0
        value can already be represented by basis
3208a1   }
61b70d };
```