一日一テーマやるやつ

一日で終わらなければ次の日も

えび

2018 年 8 月 20 日

目次

[*1] Cocke–Younger–Kasami.

1　データ構造

1.1　van Emde Boas Tree

1.1.1　2018 年 7 月 9 日

整数の集合の演算を高速に行うデータ構造．通常の演算（insertion/erasure/membership）に加え，ある値 x を超える最小の値を求める successor，その逆の predecessor を得る演算と，最小値・最大値を求める演算も行える．$[0, U-1]$ の区間の整数を全体集合とする．bound された整数の集合を扱うデータ構造であり，multiset や map として使おうとするのは険しい．

全体を陽に持つとメモリがやばいので動的に確保するが，概念としては木構造になっている．根ノードは $\sqrt{U}$ 個の子を持ち，その子は $\sqrt{\sqrt{U}}$ 個の子を持つ（以下同様）．端数が出ると厄介なので，$U = 2^{(2^n)}$ としておく．各ノードは区間の一部分を管理していて，$[0, u-1]$ を管理しているノードの i 番目（0-indexed）の子ノードは $[i\sqrt{u}, (i+1)\sqrt{u}-1]$ を管理している．実装の際には，各子ノードは $[0, \sqrt{u}-1]$ を管理する集合とし，i 番目の子ノードが値 j を持っていれば元のノードは $i\sqrt{u}+j$ を持っていると判断するようにする．

各ノードが持っているデータは次の三つである．

- そのノードが持つ集合の最小値 m・最大値 M
- 子ノードたちへのポインタ
- summary と呼ばれる集合（後述）

$[0, u-1]$ を管理するノード v の持つ summary s は $[0, \sqrt{u}-1]$ の区間を管理するノード（すなわち v の各子ノード）と同じ構造を持っていて，$i \in s$ であることと，v の i 番目の子ノードの持つ集合が空でないことが同値となるように保たせる．

ノード v が管理する区間に含まれる要素数によって，データの持ち方は異なる．要素数が 2 以下のときは m と M のみを用いて表現し，s や子ノードたちは空とする．

v が持つ集合の要素を昇順に $\{a_1, a_2, \ldots, a_n\}$ とする．$n = 0$ のときは $m > M$ など，それ以外の状態では取り得ない状態にしておく．$n = 1$ のときは $m = M = a_1$ とし，$n = 2$ のときは $m = a_1$，$M = a_2$ とする．$n \geqslant 3$ のときは $m = a_1$，$M = a_n$ として，各子ノードを用いて $\{a_2, \ldots, a_{n-1}\}$ を表現する（子ノードには a_1 および a_n は含めない）．

これのお気持ちになるとあとは簡単で，これを保つように変更したり，これを元にして探索すればよい．successor については少々頭が必要で，以下のような処理をする．クエリを x とする．

- $x \geqslant M$ や $M < m$ であれば該当する要素は存在しない．
  - x を返してエラーを示すとか，例外を投げるとかをする．
- $x < m$ なら m を返す．
- 子ノードが空なら M を返す．
- x が入るべき子ノードの持つ最大値が x を超えていれば，その子ノードに該当する要素が含まれるの

で，そのノードを探索する．
  - 該当する子ノードの添字 i は $\lfloor x/\sqrt{U} \rfloor$ となる．
- s の最大値が i 以下であれば，M を返す．
- s における i の successor j を求め，j 番目の子ノードの最小値を返す．

$U = 256$ 程度であれば，64-bit 整数 4 つを bitset として扱うことで集合を表現できるため，それを利用した．64-bit 整数 1 つで持って各階層のノード数を $\{64, 4096, 16777216, \ldots\}$ とした場合，2^{64} と噛み合わなくなるのが嫌だったのでこのようにしてある．$U = 16$ のときに特殊化する実装と $U = 256$ のときに特殊化する実装を比較したところ，後者の方が高速であった．

AOJ の Lesson の `ITP2_7_C` で verify．大きい値側から求めて逆向きに出力することで predecessor についても verify 済み．min や max は陽には verify していないが，まぁ合っている気がする（えー）．

ちなみに実測で `std::set` に勝てていません（最悪ケース時）．ケースによっては勝っているものもありますが，競プロにおいては最も時間がかかるものが指標になってしまうため．i 番目の子ノードを管理するのに `std::map` を使っているんだけど，そこを工夫できると嬉しい？

1.2 赤黒木

1.2.1 2018 年 8 月 5 日

平衡二分木で，`std::set<T>` などにも使われているデータ構造．2-3-4 木と非常によく似ている．

順序を保って平衡二分探索木として使うことで通常の演算（insertion/erasure/membership/successor/predecessor など）を高速に行えるほか，区間に対する処理を高速に行える配列として（任意の順序で）値を持っておくことも可能．以前 D. C. の online アルゴリズムでは後者として利用した．

以下の性質を保つことで，平衡を保証している．

- 各ノードは黒色または赤色である
- 根ノードは黒色である
- 赤色のノードの子は黒色である
- 根から葉までのどのパスにおいても，現れる黒色のノードの個数は等しい
- （Nil ノードは黒色である）

ノードの挿入・削除の際は，回転させたりしてこの条件を満たすように修正する[*2]．u を挿入した後，それより上のノードを修正する関数を `insert-fix-up(u)` とする．削除後の修正関数 `erase-fix-up(u)` の u がどのノードに相当するかはお絵かきが必要．子ノードへのポインタを持つときに `Node *left, *right;` とすると同じような処理を複数回書く必要があって大変なので，向きを 0 と 1 に対応づけて `Node *children[2];` としておくと実装が楽．

[*2] 詳細は略．お絵かきすると気持ちがわかってくるはず．

また，あるノード v を基準として左右に分ける $\texttt{split}(v)$ や，あるノード v を仲介して別の赤黒木 T とくっつける $\texttt{merge}(T, v)$ もできる．赤黒木同士を直接くっつけたい場合は T の最初のノードを外し，それを v として merge すればよい．

$\texttt{merge}(T, v)$ ですることは，以下の通りである．T の方が自身より低いとして話を進めるが，逆の場合（等しい場合を含む）は適宜注意すること．

- T の根から根ノードまでの黒色ノードの個数を数える
- 自身の根から右端のノードのうち，以下を満たすノード u を見つける
  - 黒色ノードである
  - 自分（inclusive）から根までの黒色ノードの個数が上で数えた個数と等しい
- このノード u と T の根を v の子とし，u の元の親を v の親とする
- v を赤色にし，$\texttt{insert-fix-up}(v)$ をする．

$\texttt{split}(v)$ ですることは，以下の通りである．

- v の左右の部分木をそれぞれ T_L，T_R とする
- v が根になるまで以下を繰り返す
  - v の親を v_P，v の姉妹以下の部分木を T_S とする
  - v が v_P の右の子なら，T_L を merge によって $\langle T_S, v_P, T_L \rangle$ にする
  - v が v_P の左の子なら，T_R を merge によって $\langle T_R, v_P, R_S \rangle$ にする

また，自分の左の部分木のノード数を各ノードに持たせておくことで，i 番目のノードに $O(\log n)$ でアクセスすることが可能だが，各種操作をする際に適切に管理する必要がある．さらに，自分の両方の部分木のノードたちの最小の値を持たせておくことで，区間 $[i_L, i_R]$ 内の最小値を $O(\log n)$ で取得することが可能だが，これも適切に管理する必要があり，気をつける必要がある．

1.3 Crit-Bit Tree

1.3.1 2018 年 8 月 7 日

0-terminated な文字列の集合を扱う木．Trie を圧縮したものと見ることができる．

内部節点と外部節点で役割が違っていて，実装がちょっと汚くなりがち．外部節点は文字列のみを持っている．内部節点は，「左の部分木で管理される文字列たちと，右の部分木の管理される文字列たちにおいて，初めに異なるのは何 bit 目か？」という情報を持っている．実際には，初めに異なる **byte** の添字と，その添字における bit mask の形式で持っている．たとえば（1 byte の bit 数が 8 として），21 bit 目が異なる場合は，添字が 2 で bit mask が $\#04$ となる[*3]．集合内の文字列数が n のとき，節点数は合わせて $2n - 1$ となる．実装の詳細は略（これは甘え）．

[*3] 元論文ではこれのビット反転を持っていた気がするが，こっちの方が好きだったため．

節点の位置関係がいい感じになっていて，集合内の文字列のうち，ある prefix を持つ文字列を列挙することが可能．

1.4 部分永続配列

1.4.1 2018 年 8 月 7 日

バージョン管理ができる配列みたいなもの．更新は最新バージョンに対してのみだが，過去のバージョンにもアクセスできる．

連想配列の配列 $\langle a_1, a_2, \dots \rangle$ を持っておく．バージョン t で添字 i の値を v にするとき，a_i に対応付け $t \mapsto v$ を追加する．前回の更新バージョン t' と今回の更新バージョン t に対して $t' \leqslant t$ が成り立つように確認する．

バージョン t での添字 i の値は，a_i において $t' \leqslant t$ を満たす最大の t' に対応付けられた値である．これは（当然）二分探索で求められる．まず a_i 内の最大のキー t_{max} とクエリ t を比較して，$t_{max} \leqslant t$ なら $a_i(t_{max})$ を返すようにすると，最新バージョンについては $O(1)$ で答えられる．また，番兵として $0 \mapsto v_{init}$ をすべての a_i に入れておくと実装が楽かも．

`std::map<size_t, T>` を使うなら（バージョンが単調非減少なので）`.emplace_hint(.end(), t, v)` をすると更新が $O(1)$ になってくれそう．自分で実装した時はそれに気付かなかったので `std::vector<std::pair<size_t, T>>` でやってしまった．

1.5 部分永続 Union-Find

1.5.1 2018 年 8 月 7 日

Union-find はただの配列があれば実装できるので，部分永続 Union-find も当然部分永続配列があれば実装できる．

ただし，部分永続配列は過去のバージョンへの更新をしないことになっているので，経路圧縮による高速化はできないはず．いわゆるマージテクによる高速化をするとよい．

1.6 Quick-Find

1.6.1 2018 年 8 月 7 日

`find`(u, v) に対して $O(1)$ で答えられる Union-find．頂点 $\langle v_1, v_2, \dots, v_N \rangle$ があるとし，それら各々がグループ $\langle g_1, g_2, \dots, g_M \rangle$ のいずれかに属している状況を考え，以下の二つの対応づけを管理する．

- ある頂点がどのグループに属しているか？
  - $G : v_i \mapsto g_?$
- あるグループにどの頂点たちが属しているか？
  - $V : g_j \mapsto \{v_?, v_?, \dots\}$

unite(u, v) をする際は，$V(u)$ の大きさと $V(v)$ の大きさを比較し，u が大きいように適宜入れ替える．これにより，計算量を $O(\log N)$ に抑えることができる．行うべきことは以下の通りである．

- $V(G(v))$ の各要素 v' について，$v' \overset{G}{\mapsto} G(u)$ とする
- $V(G(v))$ の各要素を $V(G(u))$ に挿入する
- $V(G(v))$ を空にする

are-connected(u, v) は $G(u) = G(v)$ かどうかを答えればよい．

1.7 Weighted Quick-Find

1.7.1 2018 月 8 月 7 日

- 頂点 u から v に行くまでのコストはいくらか？（または行けないか？）
- 頂点 u から v にコスト w の辺を張る
  - v から u へはコスト $-w$ で結ばれる
    - 「重みつき」ではなく「ポテンシャルつき」とするのがよいとする立場がある
    - 私も賛成しそう

というクエリに答えるデータ構造で，Quick-find に基づく．Quick-find の説明は上にあるので省く．

各グループ j には（陽には持たない）超頂点 v_j が繋がっているとする．頂点 i がグループ j に属するとき，v_j から i へのコストを別の配列の要素 w_i に持っておく．

unite(u, v, w) を考える．$G(u)$，$G(v)$ をそれぞれ g_u，g_v とする．このとき，g_u から g_v へコスト x の辺を張ると考えることができる．

$$\mathrm{Cost}(u \to g_u) + x + \mathrm{Cost}(g_v \to v) = w$$

なので，

$$-w_u + x + w_v = w$$

となり，$x = w + w_u - w_v$ とわかる[*4]．g_v に属する各頂点にとって，超頂点からのコストが x 増えるので，対応付けを変えるのと同時に更新する．

[*4] w の要素が超頂点と頂点のどちらの向きのコストなのか？ とか，u と v のどちらからどちらに辺を張るか？ とかによって符号は変わるので，実装に合わせて適宜変えてよい．

2　グラフ理論

2.1　Dynamic Connectivity (online)

2.1.1　2018 年 7 月 10 日

Union-find に delete を追加したもの．“**This is a *much* harder problem!**”[*5]

まず森に関する D. C. を解く方法を考え，それを用いて一般のグラフに関する D. C. を解く．

2.1.2　2018 年 7 月 18 日

以下では，上記の森に対する D. C. を解く方法を示す．

まず，赤黒木を用意する．これは辺 (u, v) を要素として持ち，要素を in-order で読んだ辺の列が Euler tour を表す．処理の都合上，自己辺 (u, u) を各頂点に張っておく．赤黒木自体の理解が大変なため，すでに一週間ほど使っていてつらいが，2-3-4 木に助けてもらいながらがんばった．

ここでの赤黒木は `split` と `merge` をできる必要がある．また，値の大小関係で順を管理するのではないことに注意が必要である．`split` の際には，基準となる点から根に向かって辿り，その方向に従って左右の木にくっつけていく方法を採用した．

はじめに `reroot`(u) と呼ばれるサブルーチンを定義しておく．これは Euler tour の始点を変更する処理で，以下のようにする．

- (u, u) で `split` し，木 L と木 R に分ける
- 木 R が木 L の左に来るように `merge` する
  - 木 R と木 L の間には (u, u) が入るようにする

`link`(u, v) をする際，事前条件として u と v が既につながっていないことが求められる[*6]．頂点 u，v を表す赤黒木のノード（すなわち (u, u) と (v, v)）が属する赤黒木をそれぞれ T_u，T_v とおく．

- T_u と T_v をそれぞれ u と v で `reroot`
- T_u と T_v を (u, v) をはさんで `merge`
- T_v の末尾に (v, u) を `insert`
  - 実装によっては T_v と木 $\{(v, u)\}$ を `merge` と考えてもよい

`are-connected`(u, v) に答える際は，T_u と T_v の根が同じかどうかを見ればよい．

`cut`(u, v) の事前条件は u と v が連結していることだが，これは当たり前といえば当たり前（切れているのに切ってほしいなら no-op でよい）．

[*5] だいなみく・こねくちびちー・ぷろぶれむ

[*6] これは結構つらくて，この条件下で verify できる D. C. の問題を探すのに苦労する．そもそも存在するの？

- `reroot`(u) をして (u, v) の方が (v, u) より先に来るようにする
  - 逆でもよいが，どちらが先に来るかをはっきりさせておくと楽になる
    - これは嘘で，`reroot`(u) では (u, v) が (v, u) より先に来ることが保証できないよね
    - これも嘘で，全体が木であることを考えるとそれを保証できる
- (u, v)，(v, u) でそれぞれ `split` し，得られる木を T_J，T_K，T_L とする．
- T_J と T_L を `merge` した木が新たな T_u，T_K が新たな T_v となる．

以上で，森に関する D. C. は解決．

2.1.3 2018 年 7 月 31 日

一般のグラフ G での D. C. では，辺集合を二つに分けて考える．G 上で全域森（当然一意でないこともあるが，適当に決める）をなす辺集合を *tree edge*，その補集合を *regular edge*（または *non-tree edge*）とする．また，各辺には *specificity*（または *level*）と呼ばれる非負整数値が定められていて，specificity が k 以上の tree edge で作られる森を $\mathcal{F}_k$ とする．この森を表現するために，上で紹介したデータ構造を用いる[*7]．

`are-connected`(u, v) は簡単で，$\mathcal{F}_0$ を見ればよい．G 上での連結性は $\mathcal{F}_0$ 上での連結性と等価なため．

`link`(u, v) の手順は以下の通り．新たに辺を追加する際の specificity の初期値は 0 である．

- u と v がすでに連結なら，regular edge に (u, v) を追加して終了
- そうでない場合
  - u の属する木 T_u と v の属する木 T_v を連結する
  - tree edge に (u, v) を追加する

`cut`(u, v) の手順は以下の通り．(u, v) が regular edge なら，それを取り除いて終了．そうでなければ，ループ変数 k の初期値を (u, v) の specifity とし，$k \geqslant 0$ の間，以下を繰り返す．

- (u, v) を取り除き，u，v の属する specificity が k の木をそれぞれ T_u，T_v とする
  - T_u の頂点数の方が小さくなるように適宜入れ替える（cf. マージテク）
- T_u の各辺の specificity を 1 増やす
- T_u から出ている regular edge を順に見る
  - それが T_u と T_v を結んでいれば，それらの木を連結させて終了
  - そうでなければ，その辺の specificity を 1 増やす
- 連結させられなければ，k を 1 減らす

最後まで連結させられなければ，実際に u と v の間のパスが消えたことになる．各ステップを高速に処理するため，以下の対応づけを管理しておくとよい．

[*7] 森に関する D. C. を処理できるデータ構造ならなんでもいいはず．

- $(u, v) \mapsto$ specificity of (u, v)
- $u \mapsto$ {tree edges incident from u}
- $u \mapsto$ {regular edges incident from u}

これは Holm, de Lichtenberg および Throup による online なアルゴリズムで，log がたくさんついていて重い．Codeforces の Gym/100551-A の制約（頂点数，クエリ数ともに最大 3×10^5）には敵わなかった[*8]．

これとは別で，offline のアルゴリズムで $O(\log n)$ のものがあるので，今後それを勉強しよう．

[*8] Time limit exceeded on test 15，実質 AC では？

3　構文解析

3.1　CYK[*9]法

3.1.1　2018 年 7 月 17 日

与えられた文 S が文脈自由文法 $\mathcal{G} = [T, N, \sigma, P]$ でに含まれるかを $O(|P| \cdot |S|^3)$ で判定するアルゴリズム．生成規則の適用の仕方の復元も可能で，複数存在する場合はその旨を報告できる．

区間 DP みたいなことをする．DP テーブルの添字は「長さ」「開始位置」「非終端記号（の ID）」で，要素は適用の方法の表す整数（true/false でもよい）．すなわち，dp[i][j][k] は，S の位置 j から長さ i の部分文字列を k 番目の非終端記号から導出することが可能かどうか（あるいはその種類数）を表す．ここでは 1-indexed で記すが，もちろん 0-indexed でもできる．予め生成規則を Chomsky 標準形に直しておく必要がある．

まず，S に含まれる各文字について，その位置とそれを直接導出する生成規則を求め，それに対応する要素（長さは 1）の値を 1 にする．長さ i を 2 から順に増やしていき，その区間 $[j, j+i]$ を二分した区間 $[j, j+k-1]$ および $[j+k, j+i]$ に対応する部分文字列を見る（順に S_A，S_B，S_C とする）．S_B と S_C がそれぞれ非終端記号 b と c から導出可能で，かつ $a \to b\ c$ とするような生成規則が存在した場合，S_A は非終端記号 a から導出可能であることが分かる．すなわち dp[i][j][a] を dp[k][j][b] と dp[i − k][j + k][c] から更新する．復元したい場合は (i, j, a) から (k, b, c) に対応させる情報を持っておくとよい．

dp[|S|][1][σ] が最終的な答えである．

3.2　tsurai Parser

3.2.1　2018 年 8 月 7 日

3.2.2　2018 年 8 月 15 日（編集）

四則演算を処理する構文解析器．名前は弊学の競プロサークルの通称から．

いわゆる `<term>` と `<factor>` において書くべき処理はほぼ同じなので，演算子の優先度を引数として渡して処理する．演算子は優先度ごとに分けて持っておく．

- $\langle \mathrm{expr}_0 \rangle \coloneqq \langle \mathrm{expr}_1 \rangle \mid \langle \mathrm{expr}_0 \rangle \langle \mathrm{bop}_1 \rangle \langle \mathrm{expr}_1 \rangle$
- $\langle \mathrm{expr}_1 \rangle \coloneqq \langle \mathrm{expr}_2 \rangle \mid \langle \mathrm{expr}_1 \rangle \langle \mathrm{bop}_2 \rangle \langle \mathrm{expr}_2 \rangle$
- $\vdots$
- $\langle \mathrm{expr}_N \rangle \coloneqq \langle \mathrm{literal} \rangle \mid$ `"("` $\langle \mathrm{expr}_0 \rangle$ `")"` $\mid \langle \mathrm{uop} \rangle \langle \mathrm{expr}_N \rangle$

対応するのは上のような文法で，$\langle \mathrm{bop}_i \rangle$ は二項演算子，$\langle \mathrm{uop} \rangle$ は単項演算子にである．優先度の異なる単項演算子が出てくる場合はもう少し工夫が必要．擬似コード的なものを以下に示す[*10]．

[*9] Cocke–Younger–Kasami.
[*10] `algorithm` 的な環境を使うべきでは？

通常の算術においては，$\langle \mathrm{bop}_0 \rangle$ は + または -，$\langle \mathrm{bop}_1 \rangle$ は * または / を直接導出する非終端記号という扱いになる．

- parse(s, i, j)
  - **if** $j = N$ **then**
    - **if** s_i = "("
      - $a \leftarrow$ parse(s, ++i, 0)
      - **assert** s_i = ")"
      - ++i
      - **return** a
    - **if** s_i is digit **then**
      - parse an integer and return it
    - **if** $s_i \in \langle \mathrm{uop} \rangle$ **then**
      - $\oplus \leftarrow s_i$
      - **return** $\oplus$parse(s, ++i, N)
    - **unreachable**
  - $a \leftarrow$ parse(s, i, j + 1)
  - **while** $i \leqslant |s|$ **do**
    - $\oplus \leftarrow s_i$
    - **if** $\oplus \notin \langle \mathrm{bop}_j \rangle$ **then break**
    - $a \leftarrow a \oplus$ parse(s, ++i, j + 1)
  - **return** a

上の擬似コードでは i が 1-indexed なので注意[*11]．右結合な演算子の場合は，下から二行目を $a \leftarrow a \oplus$ parse(s, ++i, j) にするといい気がする．本当かなぁ．

[*11] 擬似コードでは添字は 1 から始めたくなりがち．非終端記号は 0 からなんだけど．

4 To-Do リスト

それ以前に学んだもので，まとめておきたいものたち．そのうち書く．

- I/O 高速化
  - これいる？
    - いらない

学びたいものたちも挙げておく．

- Starry-sky tree
  - 区間更新できるセグ木と併せて押さえたい
- Wavelet 行列
- Eer-tree
- Suffix array
- Suffix automaton
- Aho-Corasick 法
- **K**MP 法
- Z algorithm
- 桁 DP を一般化したやつ？