バケット法と平方分割

バケット法とは?

- n個の要素の「何かしらの情報」が欲しい
  ⇒ いくつかのまとまりで管理する
- このまとまりを「バケット」と呼んだりする

平方分割とは?

- バケットの大きさを$\sqrt{n}$としたときのバケット法のこと
- 区間に対する処理を$O(\sqrt{n})$に抑える

以降は平方分割をメインに話していく.

なぜ$O(\sqrt{n})$か

- $\sqrt{n}$で分割するという性質のおかげでデータの参照が$O(\sqrt{n})$になる
- やれば分かる

セグ木と何が違うのかor何が同じか

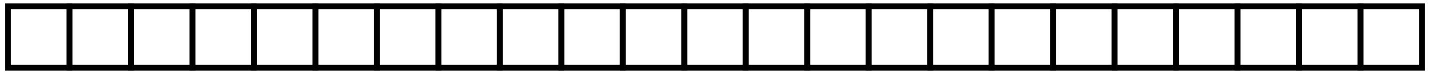
- セグ木: 元の情報を二分木として，親が子の情報をまとめる
- 平方分割: 元の情報を$\sqrt{n}$個のバケットとしてまとめる

セグ木と何が違うのかor何が同じか

- 情報をまとめる点では同じ
- まとめ方が違うだけ
    - セグ木の方が計算量的にはよい
    - 平方分割は実装が軽い

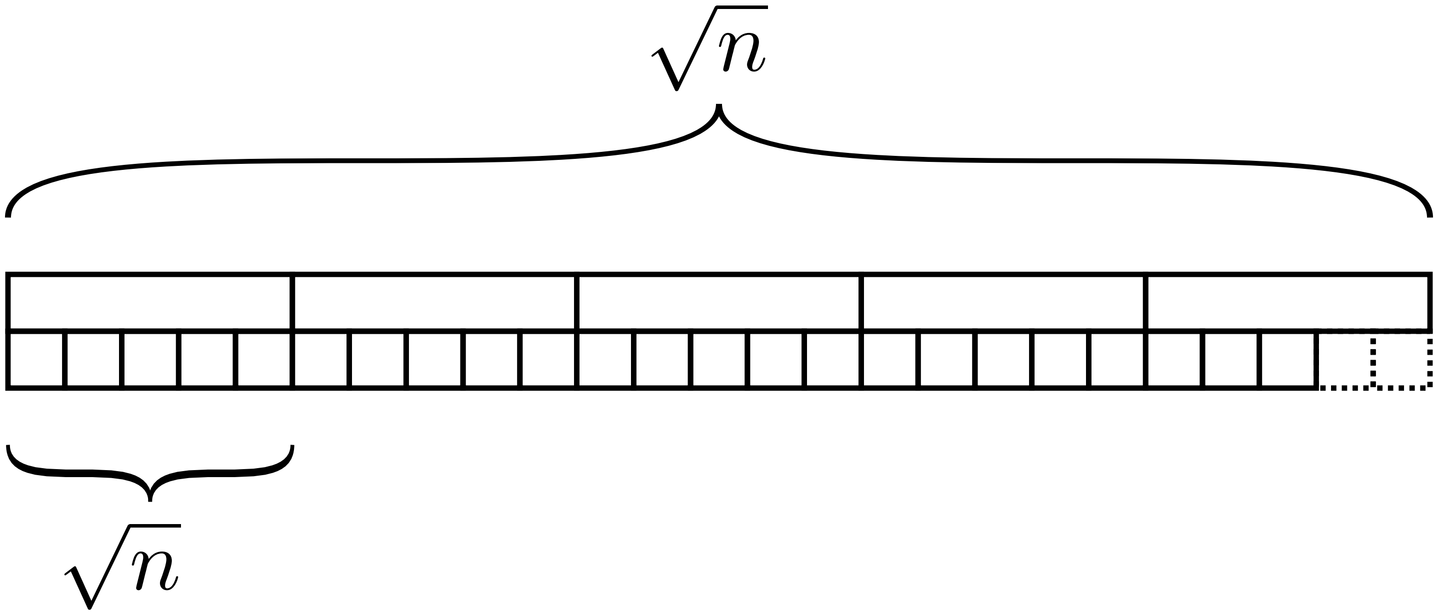
仕組み

データがあります

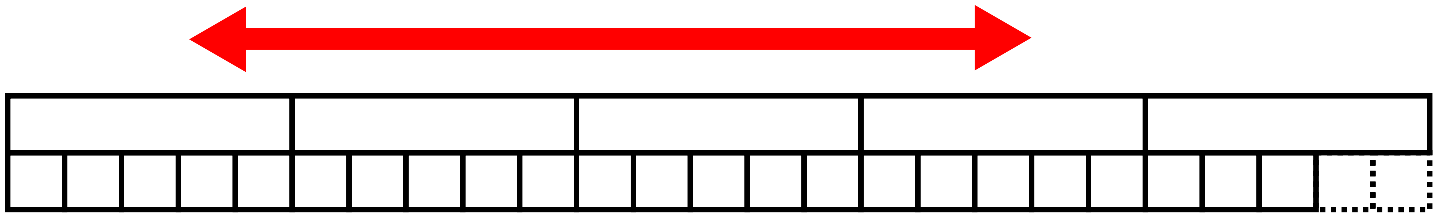


何かしらの情報を$\sqrt{n}$個のバケットにまとめます

- バケットは$\sqrt{n}$個ある
- 各バケットは$\sqrt{n}$個の要素
- わざわざ平方数に合わせなくともうまく動く

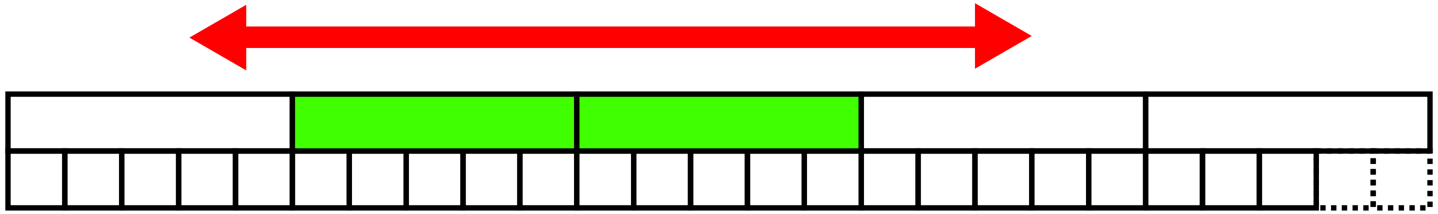


取得:例えばここの区間
の情報が欲しい



取得:ここはバケットから得られそう

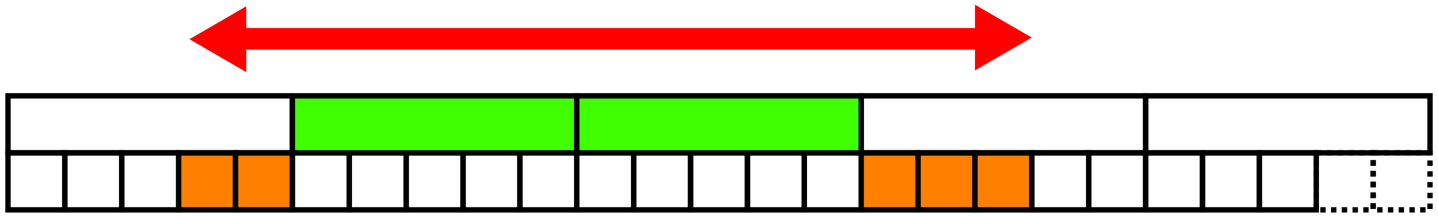
バケット全部調べる場合でも$\sqrt{n}$通りしかない



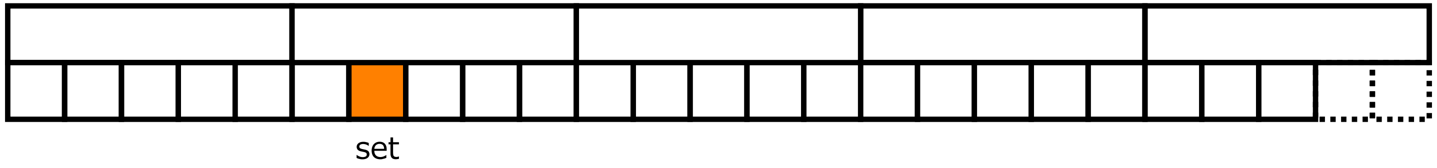
取得:中途半端なところは個別で調べれば良さそう

個別に調べる場合も

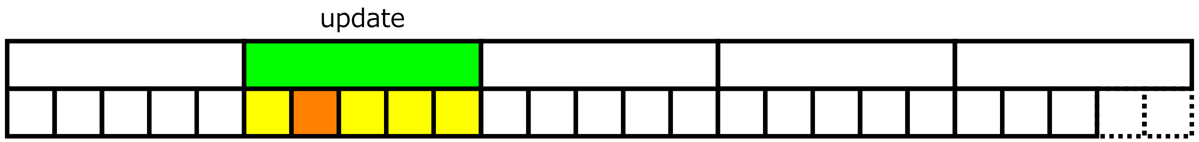
$2\sqrt{n} - 2$個しか調べない



1点更新: 例えば1点更新したとする



1点更新: バケットも更新



補足

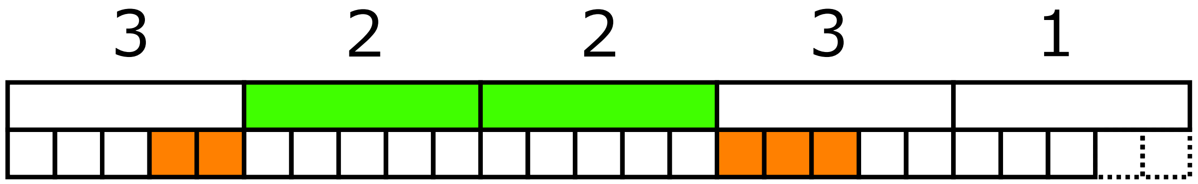
- 区間更新したい場合は工夫が必要
    - 遅延評価の力を手に入れる
    - バケットを複数持つ

実装の気持ち

取得:区間の重なり具合をみていく

バケットを左から順にみる

1. まったく重ならない: 無視
2. 完全に重なる: そのグループの値を利用
3. 部分的に重なる: 個別に計算

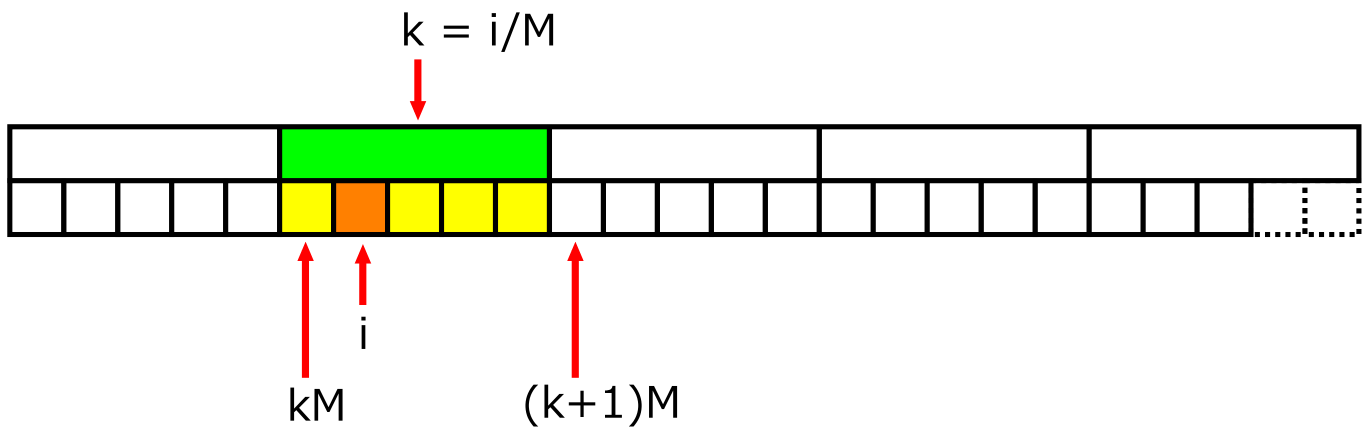


更新: 前スライドのことをそのままやる

生データとバケットの対応関係

- 要素iに対応するバケットはi/Mで計算できる
- k番目のバケットは[k*M,(k+1)*M)をまとめている

data size: N
bucket size: $M = \sqrt{N}$



バケット法の計算量 1/2

- わざわざ平方数に合わせなくともうまく動く
  ⇒ なんならバケットの大きさを定数にしてもよい
  - 例えば$\sqrt{\text{制約の最大値}}$とか

そこでバケットの大きさをbとして計算量を考えてみる

バケット法の計算量 2/2

- 1点更新: バケット全体を更新するので$O(b)$
- 区間取得:
  - 区間に完全に含まれるバケット: $O(\frac{n}{b})$個
  - 部分的に重なるものの個数: $O(b)$個

  $O(\frac{n}{b}+b)$で済みそうだと分かる
  問題によっては計算量が微妙に違う(後でやります)
  問題によってはbの値を調整しなくてはならない(後でやります)
- $b=\sqrt{n}$(平方分割)のときはどっちも$O(\sqrt{n})$になることが確かめられるね

例: RMQ

セグ木のときもやったけどもう一度やります

問題

数列 $A = \{a_0, a_1, ..., a_{n-1}\}$ に対し，次の２つの操作を行うプログラムを作成せよ．

- update(i,x): a_iをxに変更する．
- find(s,t): $a_s, a_{s+1}, ..., a_t$の最小値を出力する．
  ただし，$a_i (i = 0, 1, ..., n-1)$は，$2^{31} - 1$で初期化されているものとする．

制約

- $1 \leq n \leq 100000$
- $1 \leq q \leq 100000$
- com_iが0のとき，update(x, y):$0 \leq x_i < n, 0 \leq y_i < 2^{31} - 1$
- com_iが1のとき，find(x, y):$0 \leq x_i < n, 0 \leq y_i < n$

例題(記入スペース)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
n = 15
update(9, 2)
update(13, 21)
update(6, 28)
find(2, 3)
find(7, 9)
update(14, 45)
update(2, 36)
update(8, 45)
find(1, 13)
```

解法

初期化

```
vector<int> dat;
vector<int> bucket;
int M;
void init(int n) {
  M = 0;
  while (M * M < n) M++;
  // 生データは平方数にしたほうが扱いやすい
  dat.resize(M * M, INT_MAX);
  bucket.resize(M, INT_MAX);
}
```

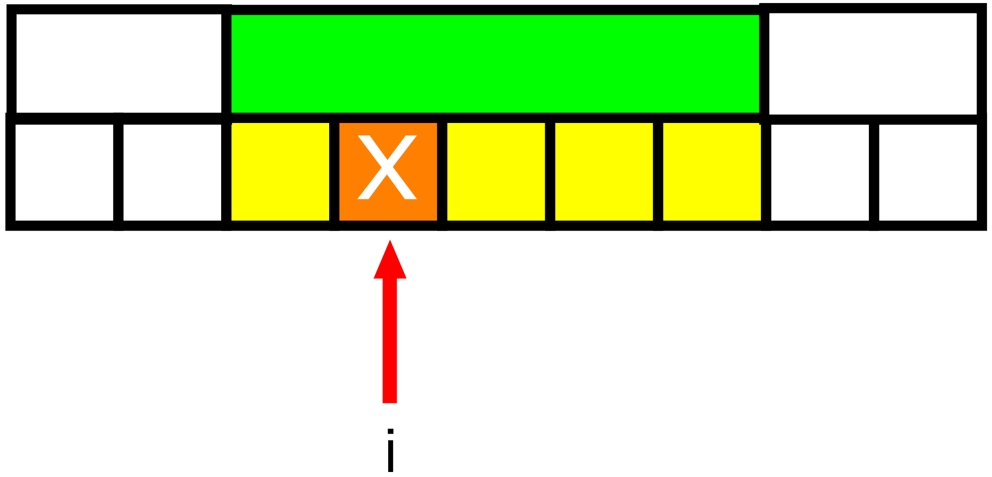
クエリ処理

- バケットには最小値を持たせる
- 生データとバケットの対応関係を思い出しながら実装しよう

data size: N
bucket size: $M = \sqrt{N}$
$k = i/M$

updateクエリ 1/2

まずi番目をxに変更



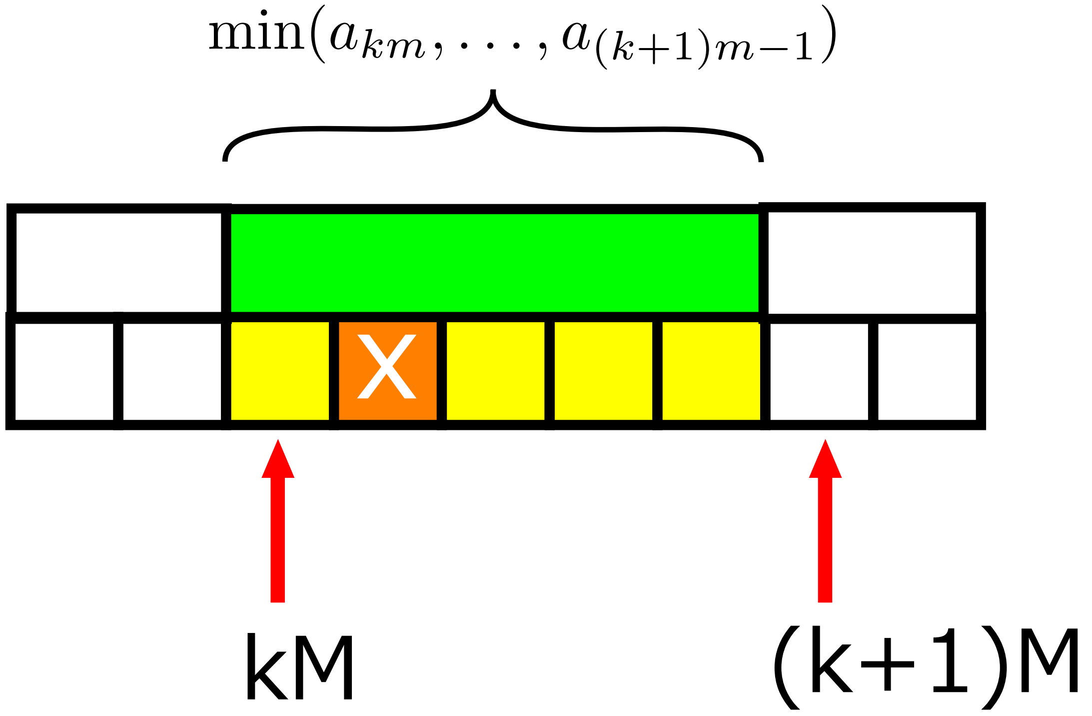
data size: N

bucket size: M = √N

k = i/M

updateクエリ 2/2

次にバケット内の全ての値
を見てバケットを更新



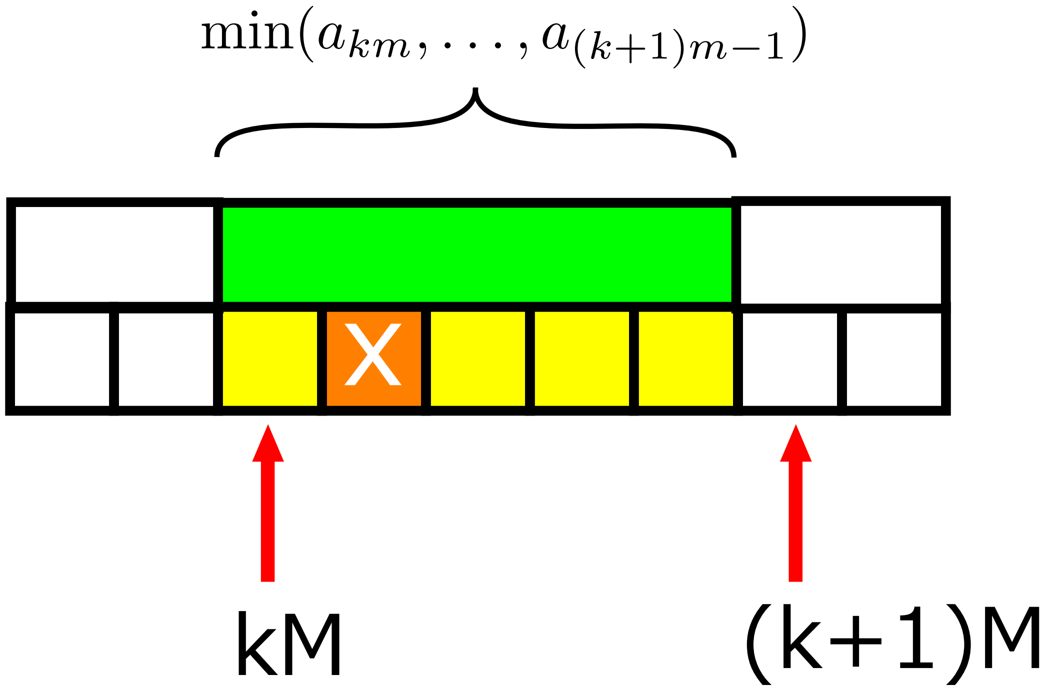
data size: N

bucket size: $M = \sqrt{N}$

$k = i/M$

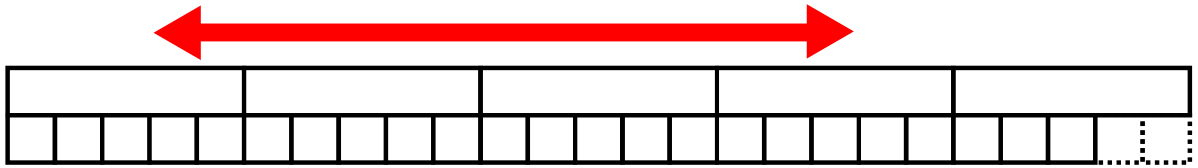
updateクエリ コード

```
void update(int idx, int x)
{
  dat[idx] = x;
  int mi = INT_MAX;
  int k = idx / M;
  for (int i = k*M; i < (k+1)*M; i++) {
    mi = min(mi, dat[i]);
  }
  bucket[k] = mi;
}
```



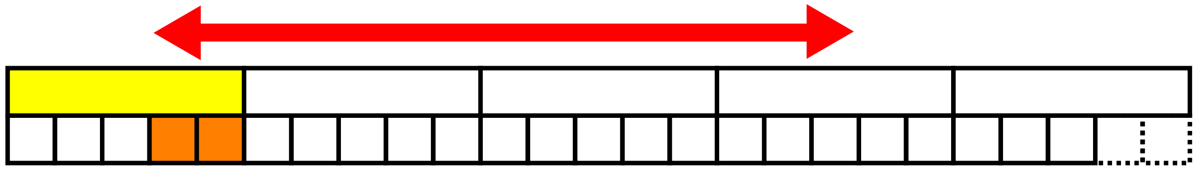
findクエリ 1/6

赤矢印の区間のminを求めたい



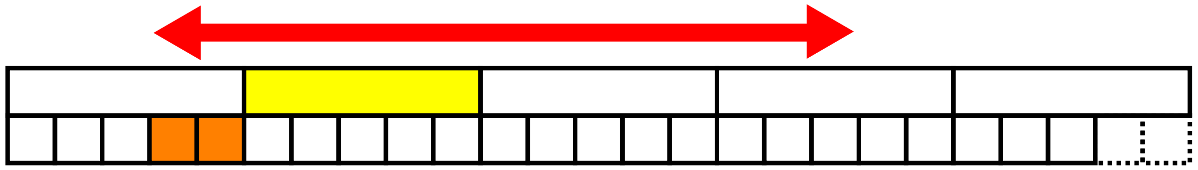
findクエリ 2/6

バケット0: 部分的に重なるので個別に計算



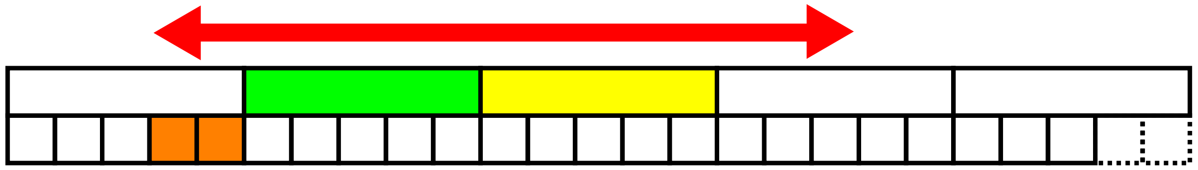
findクエリ 3/6

バケット1: 区間に包まれるのでバケットの値を利用



findクエリ 4/6

バケット2: 区間に包まれるのでバケットの値を利用

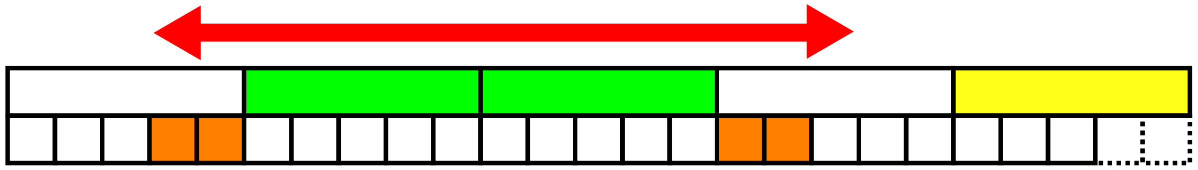


findクエリ 5/6

バケット3: 部分的に重なるので個別に計算

findクエリ 6/6

バケット4: まったく重ならないので無視

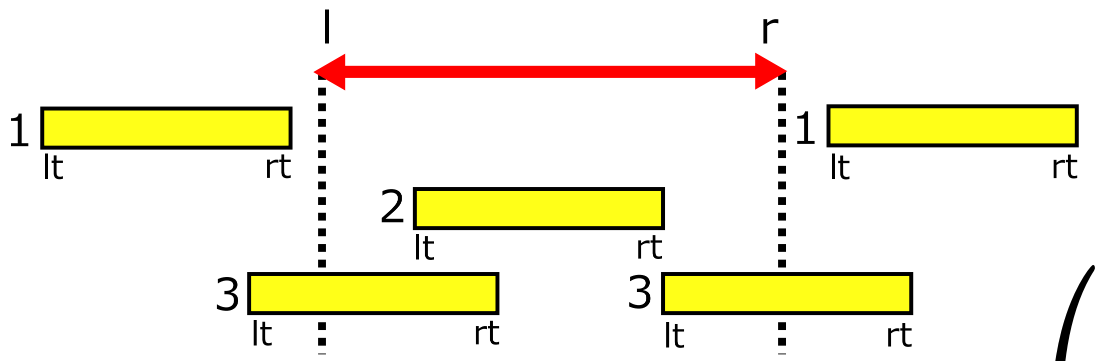


findクエリ: 区間の重なり

バケットが持つ区間と比べて

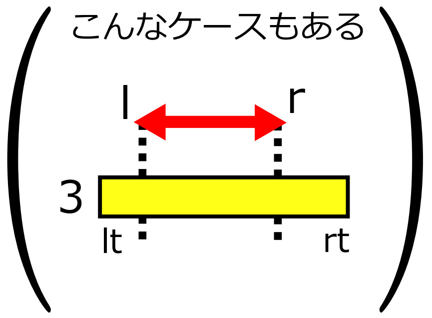
1. 重ならない: 無視
2. 包まれる: バケットの値を使う
3. それ以外: 個別に計算

3番は右のような式で表現できる．綺麗．



覚えておく:
区間[lt, rt)と区間[l, r)の共通部分は

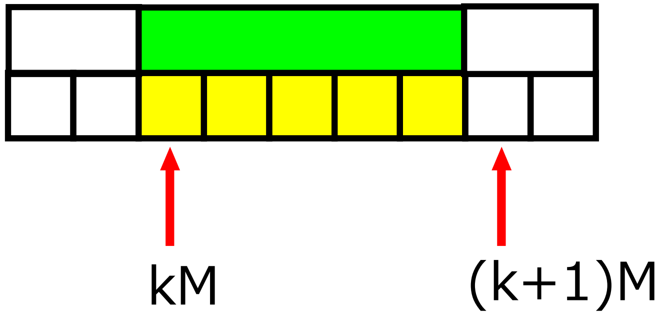
$$[\max(\mathrm{lt}, \mathrm{l}), \min(\mathrm{rt}, \mathrm{r}))$$

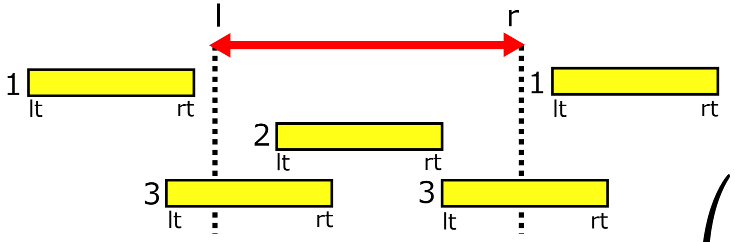


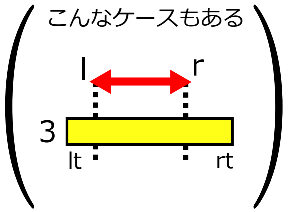
data size: N
bucket size: $M = \sqrt{N}$

findクエリ: コード

```
int find(int l, int r) {
  int ret = INT_MAX;
  for (int k = 0; k < M; k++) {
    int lt = k*M, rt = (k+1)*M;
    if (rt <= l || r <= lt) {
      continue;
    } else if (l <= lt && rt <= r) {
      ret = min(ret, bucket[k]);
    } else {
      for (int i = max(lt, l); i < min(rt, r); i++) {
        ret = min(ret, dat[i]);
      }
    }
  }
  return ret;
}
```







覚えておく:
区間[lt, rt)と区間[l, r)の共通部分は

$$[\max(\mathrm{lt}, \mathrm{l}), \min(\mathrm{rt}, \mathrm{r}))$$

例: K-th Number(POJ2104)

問題概要

Description

配列a[1...n]が与えられる
次のクエリを処理してね

- Q(i, j, k): a[i,...j]中でk番目に大きいものを求める

Input

- n: 配列のサイズ，1 ≤ n ≤ 100,000
- m: クエリ数，1 ≤ n ≤ 5,000
- |a[x]| ≤ 10^9
- i, j, k: クエリの引数，1 ≤ i ≤ j ≤ n，1 ≤ k ≤ j-i+1

Output

クエリQ(i,j,k)の答えを出力

実行時間制限

2秒

考察

どっちでもよいが0-indexedで考えることにする

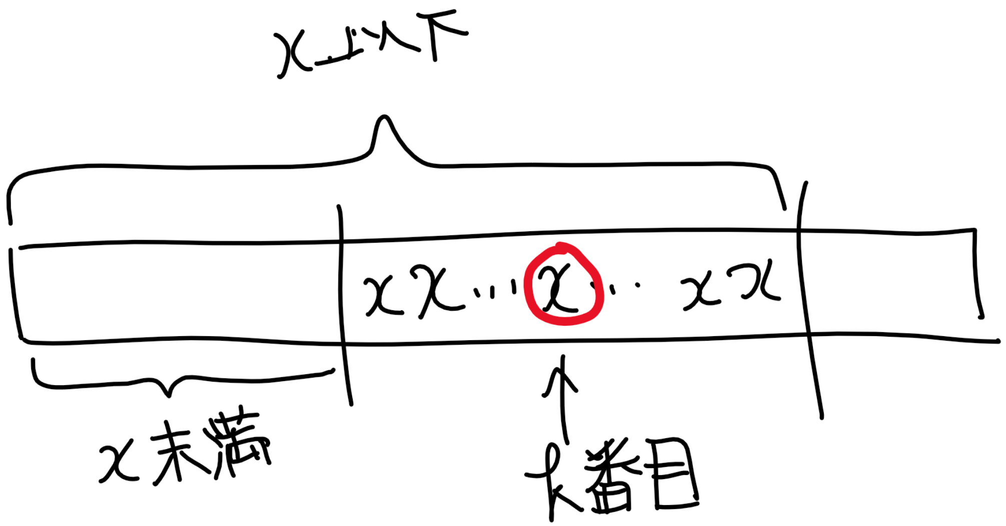
ナイーブ解法を考える

Q(i,j,k): a[i,...j]をソートしたものをbとして，b[k-1]を求める
⇒ ソートに$O(n \log n)$なので全体で$O(mn \log n)$となり間に合わない

視点を少し変える

- 「ある数xについて，それがk番目かどうか」を探索することを考える
- もちろんすべて探索することはできない
- xがk番目であるための条件について，個数に注目して考察してみよう

Sorted :



x以下 : k個以上
x未満 : k個未満

check(x) = x以下の数がk個以上か否か?

⇒ check(x) = true なる
最小のxを求めればOK

⇒ みんな大好き
にぶたん

というわけでcheck関数を実装したいが

- そのために「x以下の数」を高速に求めたいけど
- **1つ1つ**「この数はx以下かな?」とチェックしても間に合わない
- もしソートされてたら，**upper_boundで求められる**のに...

⇒ 折衷案: 平方分割

折衷案: 平方分割

- バケットからはみ出すものは1つひとつ確認する
- バケットに含まれるものはupper_boundで数える
    - バケットはあらかじめソートしておく

計算量を雑にチェック

- 前処理
  - 全部バケットをソートしておくので$O(n \log n)$
- クエリ処理
  - xについてにぶたん: $O(\log n)$
  - バケットをみる: $O(\sqrt{n})$
  - バケットについてはupper_bound: $O(\log \sqrt{n}) = O(\log n)$
  - というクエリが$O(m)$個ある

⇒ $O(n \log n + m\sqrt{n} \log^2 n)$

微妙

- $m = 5000, n = 100000$のとき, $m\sqrt{n}\log^2 n \fallingdotseq 436,204,828$

間に合わなそう…

もっとちゃんと計算量をみる

バケットの大きさをもっとちゃんと考えてあげる
バケットの大きさをbとすると，

- バケットを1つひとつみる: $O(\frac{n}{b})$
  - それぞれについてupper_bound: $O(\log b)$
- 半端なやつは1つひとつ数える: $O(b)$

全体で$O(\frac{n}{b}\log b + b)$

$$O\left(\boxed{\frac{n}{b}}\log b + \boxed{b} \cdot 1\right)$$

- $\frac{n}{b}$個のバケットに対しては $O(\log b)$
- b個の個別の要素には $O(1)$

⇒ バケットは少なめがよさそう
かといって b を大きくしすぎてもよくない。

偏らせる(厳密な平方分割とはいえなくなる)

$O(\frac{n}{b}\log b + b) = O(b)$となるように$b$を決めるのがミソ．今回は

$$b = \sqrt{n \log n}$$

くらいバケットのサイズを増やす(⇔バケットの個数を減らす)とよくて，

$$\begin{aligned} O\left(\frac{n}{b}\log b + b\right) &= O\left(\frac{n}{\sqrt{n\log n}}\log\sqrt{n\log n} + \sqrt{n\log n}\right) \\ &= O\left(\frac{\sqrt{n}}{2\sqrt{\log n}}(\log n + \log\log n) + \sqrt{n\log n}\right) \\ &= O\left(\frac{\sqrt{n}}{2\sqrt{\log n}}\log n + \sqrt{n\log n}\right) \\ &= O\left(\frac{1}{2}\sqrt{n\log n} + \sqrt{n\log n}\right) \\ &= O\left(\sqrt{n\log n}\right) \end{aligned}$$

再び計算量チェック

- 初期化時のソート
- m個のクエリ & xについてのにぶたん

⇒ 全ての計算量は

$$O(n\log n + m\log n\sqrt{n\log n}) = O(n\log n + m\sqrt{n}\log^{1.5} n)$$

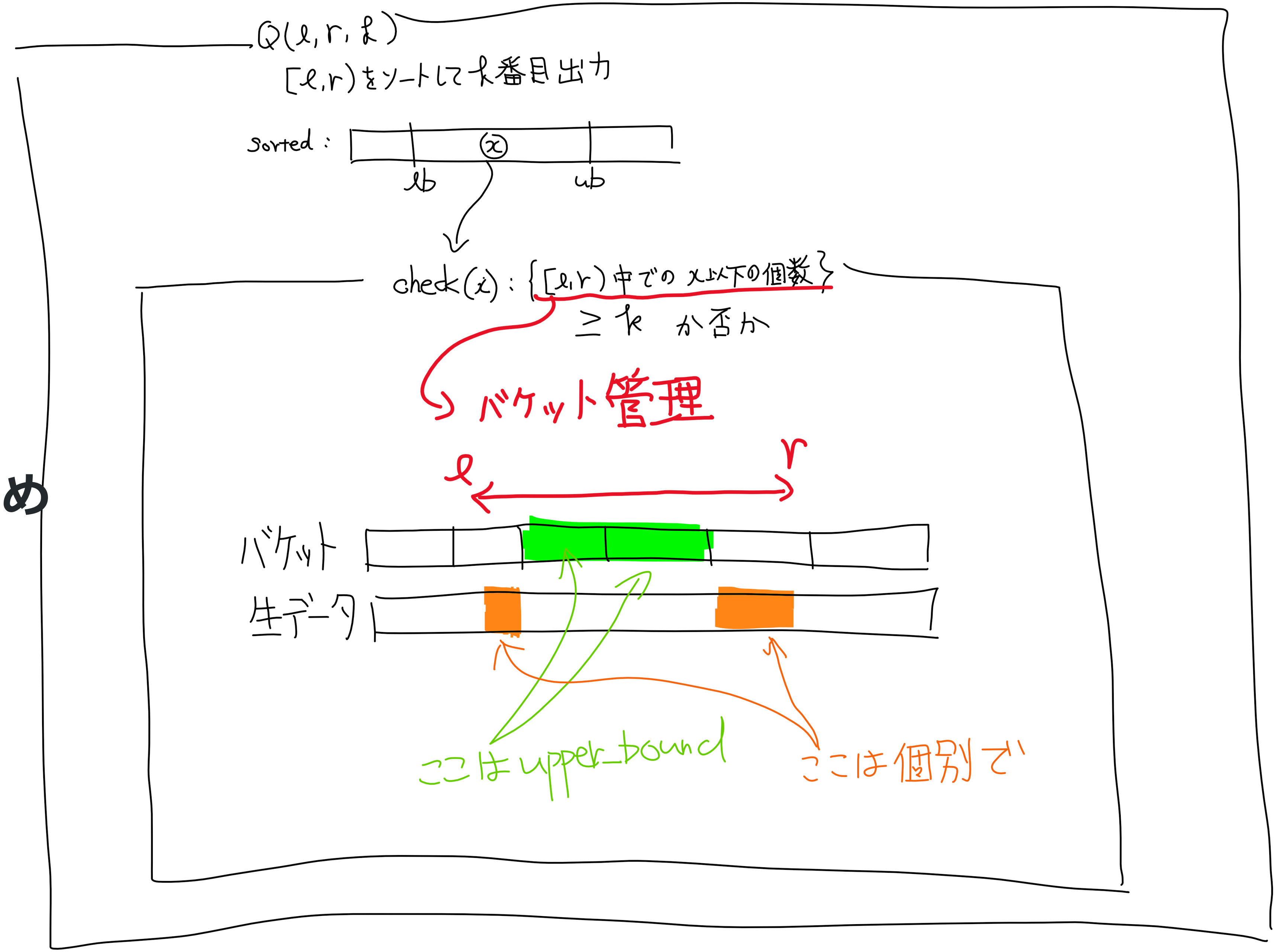
となる.

今度はどうだ

- $m = 5000, n = 100000$のとき, $m\sqrt{n}\log^{1.5} n \fallingdotseq 107,031,189$

ぎりぎり間に合う!

方針まとめ



Q(l, r, k)
[l, r)をソートしてk番目出力
sorted :
x
lb
ub
check(x) : {[l, r)中での x以下の個数} ≥ k か否か
バケット管理
l
r
バケット
生データ
ここはupper_bound
ここは個別で

実装

- bは固定する

$$b = \sqrt{10^5 \log_2 10^5} \fallingdotseq 1300$$

- 二分探索対象のxは配列の要素すべてとする．
  なのでそのために生データをソートしたデータも用意しておく．
  区間外の数もcheckすることになるが，それは絶対答えにならないので安心．
- POJだと定数倍が怖いので，なるべくC風の書き方にした

コード 1/5

```
#include <cstdio>
#include <vector>
#include <algorithm>
#define REP(i, n) for (int i = 0; i < n; i++)
#define ALL(c) c.begin(),c.end()
#define MAX_N 110000

using namespace std;

const int B = 1300;
int n, m;
int dat[MAX_N];
int num[MAX_N];
vector<int> bucket[MAX_N/B + 1];
```

コード 2/5

```
int main()
{
  scanf("%d%d", &n, &m);

  REP(i, n) scanf("%d", &dat[i]);

  init(n);

  REP(q, m) {
    int i, j, k;
    scanf("%d%d%d", &i, &j, &k);
    i--, j--;
    printf("%d\n", Query(i, j + 1, k));
  }

  return 0;
}
```

コード 3/5

```
void init(int n)
{
  REP(i, n) {
    num[i] = dat[i];
    bucket[i/B].push_back(dat[i]);
  }
  sort(num, num + n);
  for (int i = 0; i*B < n; i++) {
    sort(bucket[i].begin(), bucket[i].end());
  }
}
```

コード 4/5

```
int Query(int l, int r, int k)
{
  int lb = -1, ub = n;
  while (ub - lb > 1) {
    //printf("[%d, %d)\n", lb, ub);
    int mid = (lb + ub) / 2;
    int x = num[mid];
    if (check(l, r, k, x)) ub = mid;
    else lb = mid;
  }
  return num[ub];
}
```

コード 5/5

```
bool check(int l, int r, int K, int x)
{
  int cnt = 0;
  for (int k = 0; k*B < n; k++) {
    int lt = k*B, rt = (k+1)*B;
    if (rt <= l || r <= lt) {
      continue;
    } else if (l <= lt && rt <= r) {
      cnt += upper_bound(ALL(bucket[k]), x) - bucket[k].begin();
    } else {
      for (int i = max(lt, l); i < min(rt, r); i++) {
        if (dat[i] <= x) cnt++;
      }
    }
  }
  //printf("check(%d,%d,%d,%d): %d\n", l, r, K, x, cnt);
  return cnt >= K;
}
```

演習

SoundHound2018 本戦 A - Feel the Beet

RSQ - DSL_2_B

CF404(div2) E - Anton and Permutation

参考文献

セグメント木をあきらめた人のための平方分割