Advanced Algorithms for Programming Contests
Lecture 9. Interval data structures

Bakhodir Ashirmatov,
Thilo Stier,
Philip Schär

University of Göttingen
Institute of Computer Science

19.06.2019

Overview

General
- Interval query task
- Basic ideas

Sparse tables

Fenwick trees
- Normal Fenwick tree
- Range update Fenwick tree
- Fenwick tree 2D

Interval tree

Interval query task

- Given an array of numbers $a_0, ..., a_n$
- We consider functions on integer intervals $[i, j]$ $(0 \leq i \leq j \leq n)$, that depend on $a_i, ..., a_j$.
    1. $GET(i, j)$ find value on interval $[i, j]$
    2. $UPDATE(i)$ update a_i
    3. $UPDATE(i, j)$ update $a_i, ..., a_j$
- Typical *GET* functions: *SUM*, *MIN*, *MAX*

Basic ideas

- Plain array: $GET(i, j)$ in $O(n)$ and $UPDATE(i)$ in $O(1)$
- Optimize queries with precalculations.
- Consider $GET = SUM$:
  - Partial sum array: $SUM(i, j)$ in $O(1)$ and $UPDATE(i)$ in $O(n)$
  - Difference array: $SUM(i, j)$ in $O(n)$ and $INCREMENT(i, j)$ is $O(1)$

Sparse tables

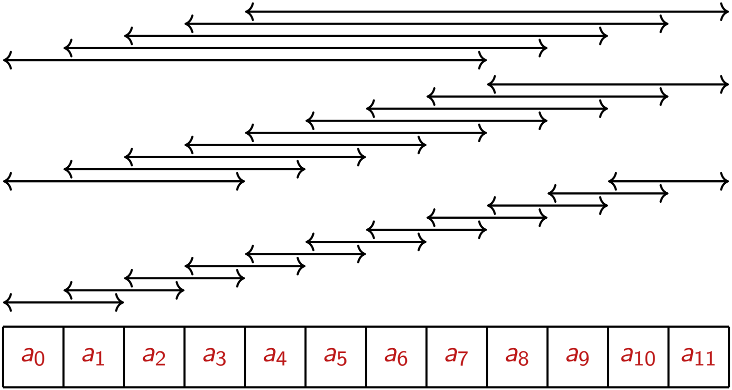
Problem

- Answer many queries $MIN(l, r)$ or $MAX(l, r)$

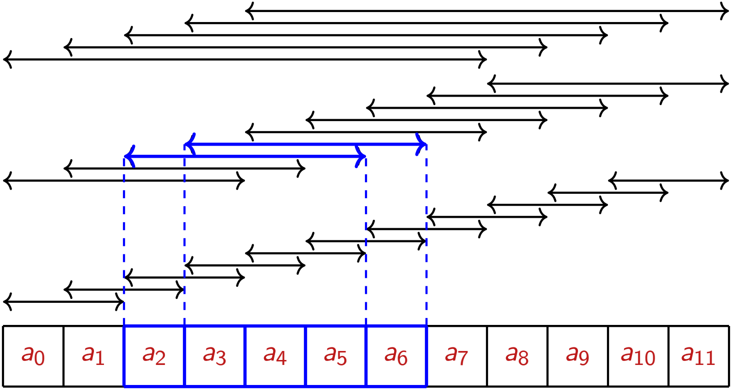
Idea

- Precalculate minima of sub-arrays.
- Calculate $MIN(l, r)$ as minimum of values of intervals that cover $[l, r]$.
- Overlappings don't matter, since $\min(\min(a, b)) = \min(a, b)$.
- How to choose sub-arrays to precalc?
- Best: only two intervals needed to cover $[l, r]$.

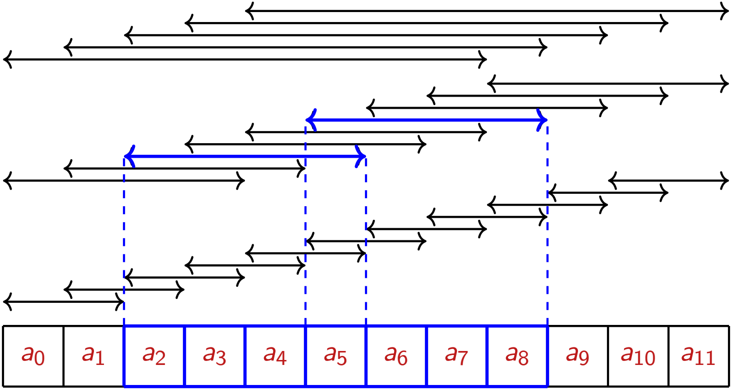
Sparse tables



Sparse tables



Sparse tables



Sparse tables

Algorithm

- $ST[k][i]$ is minimum on interval $[i, i + 2^k - 1]$ (length 2^k).
- Especially $ST[0][i] = a_i$ (initialized).
- Recursive: $ST[k][i] = \min(ST[k-1][i], ST[k-1][i + 2^{k-1}])$.
- $MIN(l, r) = \min\{ST[\lfloor \log_2(r - l + 1) \rfloor][l],$
  $ST[\lfloor \log_2(r - l + 1) \rfloor][r - 2^{\lfloor \log_2(r-l+1) \rfloor} + 1]\}$

Complexity

- $MIN(l, r)$ or $MAX(l, r)$ in $O(1)$.
- Precalc and memory $O(n \log(n))$.

Sparse tables

```
const int MAXN = 1e6, MAXLOG = 19, INF = 1e9;
int sparse[MAXLOG][MAXN];
int logsize[MAXN];

void build(int n) {
  int c = 2;
  for (int k = 1; l < MAXLOG; k++) {
    for (int i = 0; i + (1 << k) < n; i++)
      sparse[k][i] = min(sparse[k-1][i], sparse[k-1][i + (1<<(k-1))]);
    while (c <= min(2 << k, n))
      logsize[c++] = k;
  }
}

void get(int l, int r) {
  if (l > r)
    return INF;
  int lg = logsize[r - l + 1];
  return min(sparse[lg][l], sparse[lg][r - (1<<lg) + 1]);
}
```

Fenwick tree

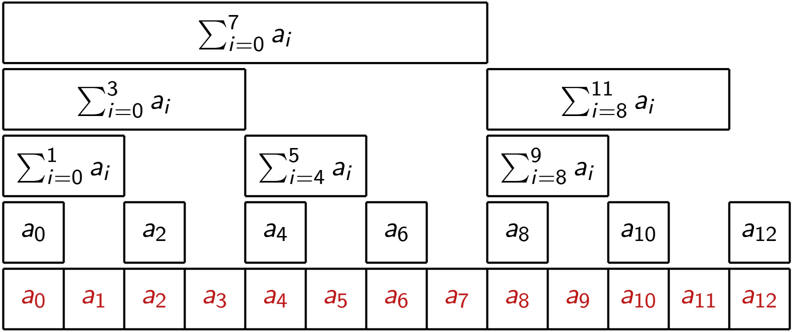
Problem

- Efficiently calculate (prefix-) sums on changing array.
- Efficiently modify values in array.

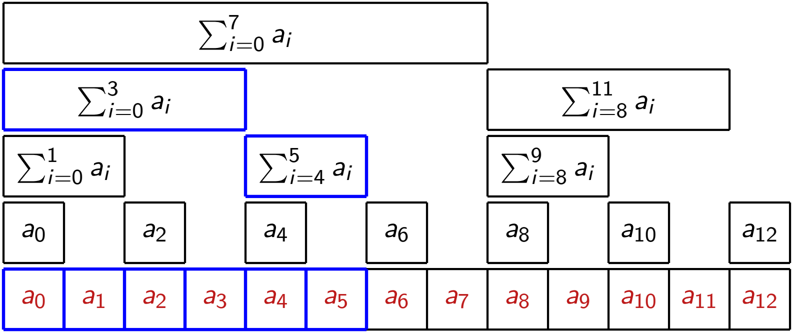
Idea

- Store partial sums of length 2^n starting at $k \cdot 2^{n+1}$.
- Use binary representation to quickly find correct precalculated sums.

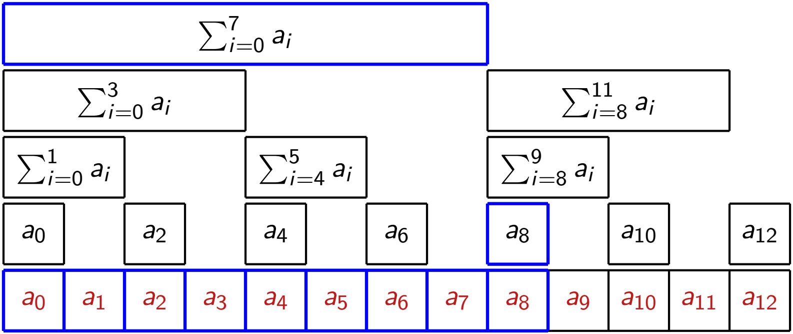
Fenwick tree



Fenwick tree

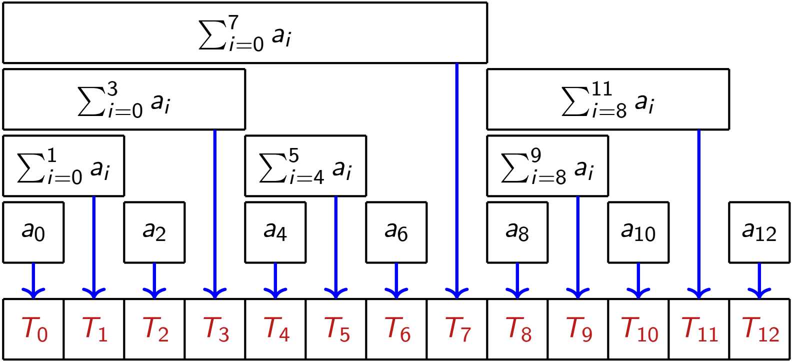


Fenwick tree



Fenwick tree

- Representation in array: See analogy to binary numbers.
- Store each partial sum at the index, where it ends (this is unique).



Fenwick tree

A little excursus on bitmagic

- Length of interval for T_n is two to the number k of consecutive ones at the end of n.
- To find the beginning, we need to remove all consecutive ones (therefore decreasing the value by $2^k - 1$).
- Adding 1 flips all bits up to the lowest order zero.
- So the beginning of the interval ending at n is: $F(n) := n\&(n+1)$
- Find the interval containing $[F(n), n]$: We have to flip the lowest zero.
- Obtained by $n|(n+1)$: leading ones untouched, first zero set to one.

Fenwick tree

Algorithm

- Let $t[n] = \sum_{F(n)}^{n} a_i$ with $F(n) = n\&(n+1)$ (delete trailing ones).
- $SUM(0, n) = t[n] + SUM(0, F(n) - 1)$.
- $SUM(l, r) = SUM(0, r) - SUM(0, l - 1)$.
- $INCREMENT(n)$ updates $t[n]$ and calls $INCREMENT(n|(n+1))$.

Complexity

- Calculate $SUM(l, r)$ in $O(\log(n))$.
- Calculate $INCREMENT(i)$ in $O(\log(n))$.
- Memory $O(n)$.

Fenwick tree

```
int t[MAXN];

void init(int n){
  for (int i = 0; i < n; i++)
    inc(i, a[i]);
}

int sum(int r) {
  int result = 0;
  for (; r >= 0; r = (r & (r+1)) - 1)
    result += t[r];
  return result;
}

void inc(int i, int delta) {
  for (; i < n; i = (i | (i+1)))
    t[i] += delta;
}

int sum(int l, int r) {
  return sum(r) - sum(l-1);
}
```

Range update Fenwick tree

Problem

- Efficiently increment ranges.
- Efficiently get values at single positions.

Idea

- Use difference array $d[i] = a_i - a_{i-1}$ (with $a_{-1} := 0$).
- With that $a_n = \sum_{i=0}^{n} d[i]$.
- Increment on range $[l, r]$ only changes $d[l]$ and $d[r+1]$:
  - For $i < l$ or $i > r + 1$: $d[i] = a_i + a_{i-1}$ unchanged.
  - For $i \in [l + 1, r]$: $d[i] = (a_i + v) - (a_{i-1} + v) = a_i - a_{i-1}$ unchanged.
  - $d[l] = (a_i + v) - a_{i-1}$ incremented by v.
  - $d[r + 1] = a_i - (a_{i-1} + v)$ decremented by v.

Range update Fenwick tree

Algorithm

- Transform $d[i] = a_i - a_{i-1}$ at init.
- Implement $GET(n)$ as $SUM(n)$ of default Fenwick.
- Implement $INC(l, r, v)$ as $INC(l, v)$, $INC(r + 1, -v)$.

Complexity

- Calculate $GET(i)$ in $O(\log(n))$.
- Calculate $INCREMENT(l, r)$ in $O(\log(n))$.
- Memory $O(n)$.

Range update Fenwick tree

```
int t[MAXN];

void init (int n){
  inc(0, a[0]);
  for (int i = 1; i < n; i++)
    inc(i, a[i] - a[i-1]);
}

int get(int i) {
  int result = 0;
  for (; i >= 0; i = (i & (i+1)) - 1)
    result += t[i];
  return result;
}

void inc(int l, int r, int delta) {
  for (; l < n; l = (l | (l+1)))
    t[l] += delta;
  for (r++; r < n; r = (r | (r+1)))
    t[r] -= delta;
}
```

Fenwick tree 2D

Problem

- Calculate sums on rectangles $[x_1, x_2] \times [y_1, y_2]$ on changing 2D array.
- Change array at single points (x, y).

Idea

- Combine two Fenwick trees: $t[x][y] = \sum_{i=F(x)}^{x} \sum_{j=F(y)}^{y} a_{ij}$.
- Calculate rectangular sum with inclusion-exclusion principle.

Complexity

- $SUM(x_1, y_1, x_2, y_2)$ and $INC(x, y, v)$ in $O(\log^2(n))$
- Memory $O(n^2)$.

Fenwick tree 2D

```
int t[MAXN][MAXN];

int sum(int x, int y) {
  int result = 0;
  for (int i = x; i >= 0; i = (i & (i+1)) - 1)
    for (int j = y; j >= 0; j = (j & (j+1)) - 1)
      result += t[i][j];
  return result;
}

int sum(int x_1, int y_1, int x_2, int y_2) {
  return sum(x_2, y_2) - sum(x_1, y_2) - sum(x_2, y_1) + sum(x_1, y_1);
}

void inc(int x, int y, int delta) {
  for (int i = x; i < n; i = (i | (i+1)))
    for (int j = y; j < m; j = (j | (j+1)))
      t[i][j] += delta;
}
```

Interval tree

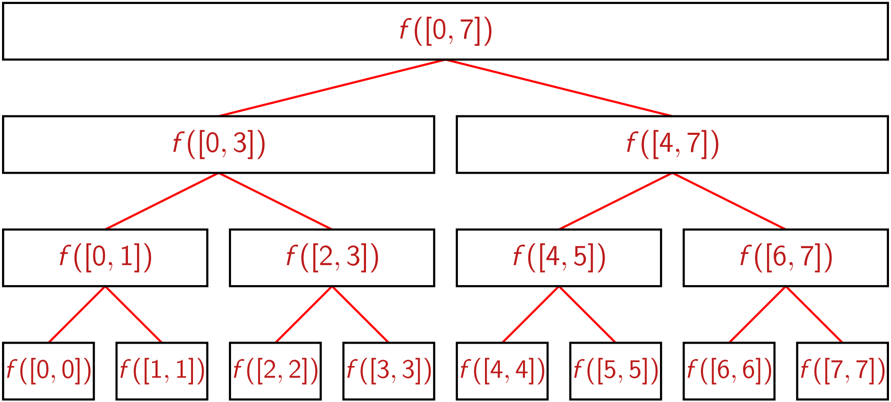
Problem

- Calculate *MIN*, *MAX* (or other) on changing array.

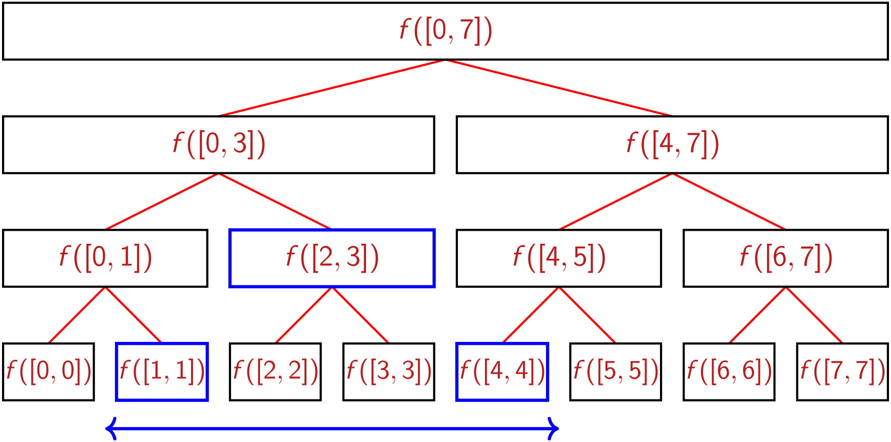
Idea

- Generalize Fenwick tree:
- Store value on intervals of length 2^n starting at $k \cdot 2^n$.
- Any interval is disjoint union of such intervals.

Interval tree



Interval tree



Interval tree

Algorithm

- Store tree in array: 1 is root, $2n$ and $2n+1$ are children of n.
- Interval $[t_l, t_r]$ is divided into $\left[t_l, \lfloor\frac{t_l+t_r}{2}\rfloor\right]$ and $\left[\lfloor\frac{t_l+t_r}{2}\rfloor + 1, t_r\right]$
- Needed array size not be larger than $4n$ ($2n$ for powers of 2).
- $t[n] = \min\{t[2 \cdot n], t[2 \cdot n + 1]\}$ (or $\max$, sum, etc.)

Complexity

- Build is $O(n)$.
- Get and update are $O(\log(n))$
- Memory $O(n)$

Interval tree

- Recursive build: build tree rooted at v, which represents interval $[t_l, t_r]$ based on array a.
  - Trivial for intervals with only one element.
  - Build left and right subtree and calculate value for v based on those.

```
int n, t[4*MAXN];

void build(int a[], int v = 1, int tl = 0, int tr = n) {
  if (tl == tr)
    t[v] = a[tl];
  else {
    int tm = (tl + tr) / 2;
    build(a, 2*v, tl, tm);
    build(a, 2*v + 1, tm + 1, tr);
    t[v] = min(t[2*v], t[2*v + 1]);
  }
}
```

Interval tree

- Recursive query: Find value on $[l, r]$ in tree rooted at v, which represents $[t_l, t_r]$.
  - Can't find it, if the interval is empty $\rightarrow$ return a neutral value.
  - If $l = t_l$ and $r = t_r$ we have the value precalculated.
  - Find values in left and right subtree and combine.

```
int get(int l, int r, int v = 1, int tl = 0, int tr = n) {
  if (l > r)
    return INF;
  if (l == tl && r == tr)
    return t[v];
  int tm = (tl + tr) / 2;
  return min(get(l, min(r, tm), 2*v, tl, tm),
             get(max(l, tm + 1), r, 2*v + 1, tm + 1, tr));
}
```

Interval tree

- Recursive update: Set element at *pos* to *new_val* in tree rooted at *v*, which represents $[t_l, t_r]$.
  - Trivial for intervals with only one element.
  - Look which subtree contains *pos* and update that one.
  - Update value of *v*

```
void update(int pos, int new_val, int v = 1, int tl = 0, int tr = n) {
  if (tl == tr)
    t[v] = new_val;
  else {
    int tm = (tl + tr) / 2;
    if (pos <= tm)
      update(pos, new_val, 2*v, tl, tm);
    else
      update(pos, new_val, 2*v + 1, tm + 1, tr);
    t[v] = min(t[2*v], t[2*v + 1]);
  }
}
```

Do your homework!