Data Structure (IV)

Daniel Hsieh {QwertyPi}

2024-06-01

What we will cover today

1. Designing data structures for higher dimensional / complex data
   - Nesting data structures (2d Fenwick Trees and more)
2. Advanced Divide & Conquer techniques
3. Time Travelling with persistent data structures!
4. Practice problems :)

Tasks to cover together

Higher Dimensional / More Complex Data Structures:

M1952 – Rapping in HKOI

AP181 – New Home

T192 – Colorful Strip

I1813 - Werewolf

- If you can solve all of them, feel free to stop listening :)
- (I recommend attempting from **top to bottom** on each slide)

Tasks to cover together

Advanced Divide & Conquer Techniques

APIO 2019 T3 – Street Lamps
M1953 – Sightseeing Trail
M1962 – Planar Game

- If you can solve all of them, feel free to stop listening :)
- (I recommend attempting from **top to bottom** on each slide)

Tasks to cover together

Persistent Data Structures

M1842 – Another RMQ

APIO 2019 – Land of Rainbow Gold

NOI 2018 – 归程

- If you can solve all of them, feel free to stop listening :)
- (I recommend attempting from **top to bottom** on each slide)

What you already know...

Range Queries

- Given an array of a[1...n]
- Support some operations
  - **Update** : Given i, v → set a[i] to v
  - **Sum** : Given l, r → find s = a[l] + ... + a[r]

What you already know…

Range Queries

- Given an array of a[1…n]
- Support some operations
  - **Update** : Given i, v → set a[i] to a[i] + v
  - **Sum** : Given l, r → find s = a[l] + … + a[r]

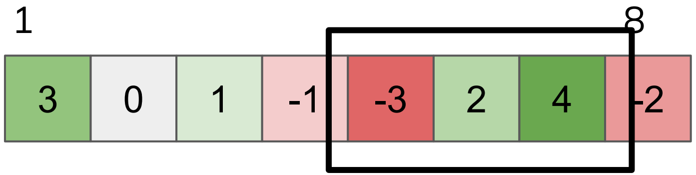
- If you are here, you should be **very very familiar** with this

What you already know…

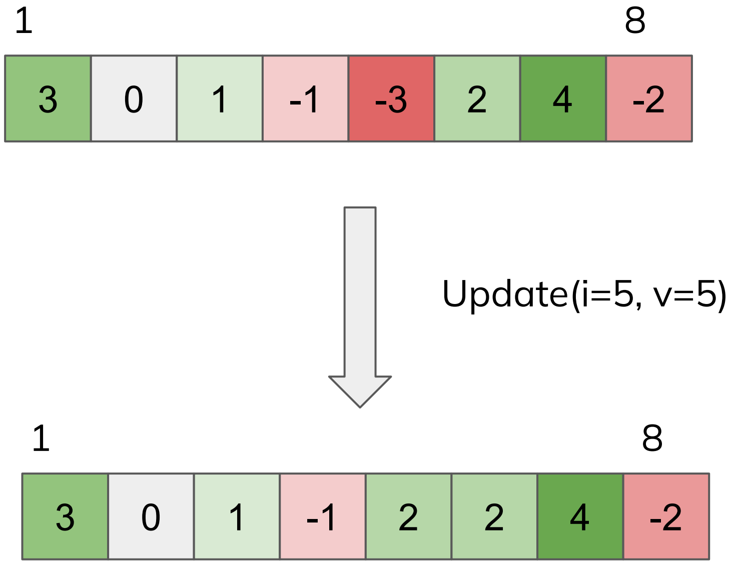
Range Queries

```
int a[n+5];
int range_sum(int l, int r) {
    int sum = 0;
    for (int i = l; i <= r; ++i) {
        sum = sum + a[i];
    }
    return sum;
}

void update(int i, int v) {
    a[i] = v;
    return;
}
```

A Graphical Representation



Sum(l=5, r=7) = -3 + 2 + 4 = 3



Solution

- Many solutions: Sparse Table, Segment Tree, **Fenwick**, BST, ...

- Let's focus on **Fenwick Tree**
  - **Update(i, v):** FenwickTree::update(i, v)
  - **Sum(l, r):** FenwickTree::query(r) - FenwickTree::query(l-1)

```cpp
template<typename T>
struct FenwickTree {
    void update(int index, T value);
    T query(int index);
    int sum(int l, int r) { return query(r) - query(l-1); }
};
```

2d Orthogonal Range Queries

- Given grid a[1...n, 1...m]
- Support some operations
  - **Update :** Given r, c, v → set a[r, c] to a[r, c] + v
  - **Sum :** Given $r_1, c_1\ r_2, c_2$
    → find sum in rectangle enclosed by $(r_1, c_1), (r_2, c_2)$

c=1 8

3	0	1	-1	-3	2	4	-2
3	0	1	-1	-3	2	4	-2
3	0	1	-1	-3	2	4	-2

Sum(r1=1, c1=5, r2=2,c2=7)
= -3 + 2 + 4 + -3 + 2 + 4
= 6

<O(1), O(n^2)> solution

For simplicity, assume **n = m**

Notation:

- <O(p), O(q)> – O(p) for update, O(q) for sum query
- You now have **10** seconds to come up with a <O(1), O(n^2)> solution :)

<O(log n), O(n log n)> solution

- We have: <O(log n), O(log n)> solution for each row
- Construct one Fenwick Tree for each row:
    - Build: fenwick_tree = FenwickTree<int, n>[n];
    - Query(r1, c1, r2, c2):
        - Iterate i=1 from r1 to r2 and add fenwick_tree[i]::sum(c1, c2)
        - Time: n * O(log n) = O(n log n)
    - Update(r, c, v):
        - Call fenwick_tree[r]::update(c, v)
        - Time: O(log n)

<O(log n), O(n log n)> solution

Does this ring a bell?

```
FenwickTree<int> tree[n+5];

int orthogonal_range_sum(int r1, int c1, int r2, int c2) {
    int sum = 0;
    for (int i = r1; i <= r2; ++i) {
        sum = sum + tree[i].sum(c1, c2);
    }
    return sum;
}

void update(int r, int c, int v) {
    return tree[r].update(c, v);
}
```

<O(log n), O(n log n)> solution

Reminder: 1d query

```
int a[n+5];
int range_sum(int l, int r) {
    int sum = 0;
    for (int i = l; i <= r; ++i) {
        sum = sum + a[i];
    }
    return sum;
}

void update(int i, int v) {
    a[i] = v;
    return;
}
```

A slower <O(log^2 n), O(n log n)> solution

Let’s try a Fenwick Tree of Fenwick Tree!

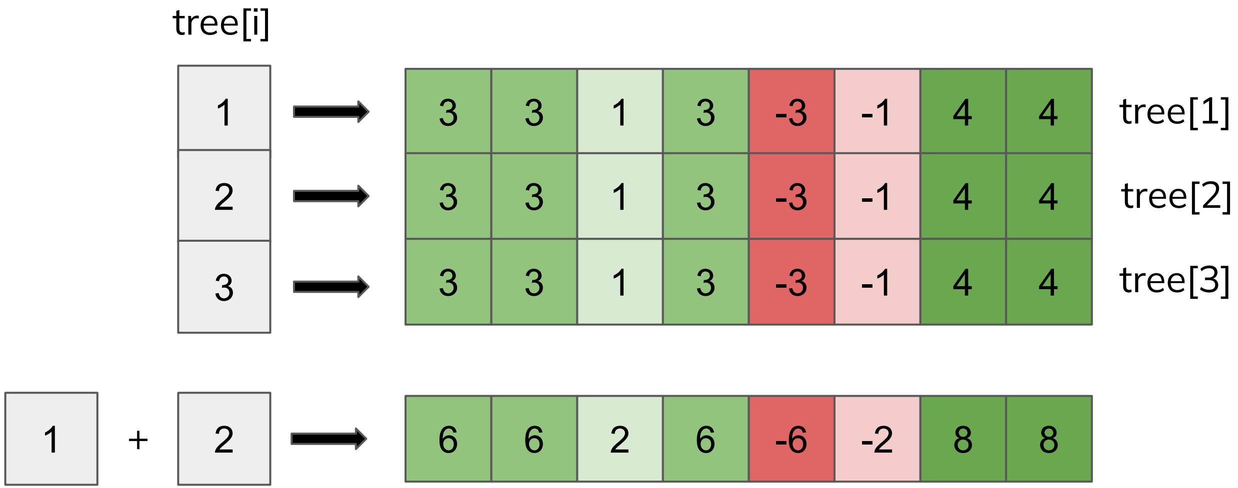
- List of Fenwick Tree for previous examples

tree[i]										
1	→	3	3	1	3	-3	-1	4	4	tree[1]
2	→	3	3	1	3	-3	-1	4	4	tree[2]
3	→	3	3	1	3	-3	-1	4	4	tree[3]

A slower <O(log^2 n), O(n log n)> solution

Sum(r1=1, c1=5, r2=2, c2=7)

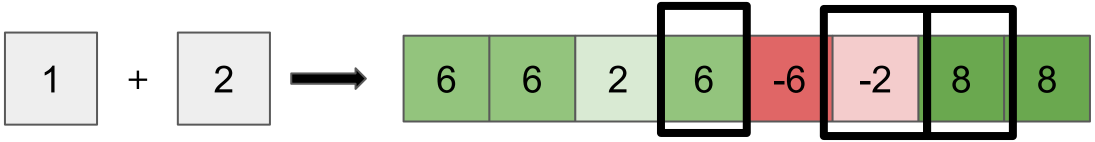
1. Find $\boxed{1} + \boxed{2}$



A slower <O(log^2 n), O(n log n)> solution

Sum(r1=1, c1=5, r2=2, c2=7)

2. Query 1 + 2



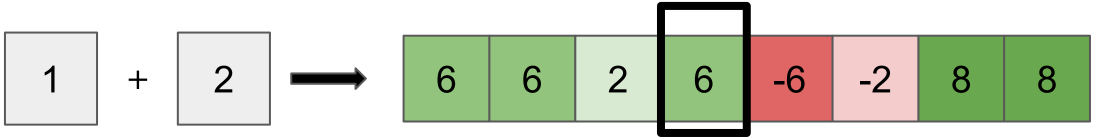
Query(7) = 8 + -2 + 6 = 12

I’m not going to show how I get these indexes

A slower <O(log^2 n), O(n log n)> solution

Sum(r1=1, c1=5, r2=2, c2=7)

2. Query $\boxed{1}$ + $\boxed{2}$



Query(7) = 8 + -2 + 6 = 12
Query(5-1) = Query(4) = 6

A slower <O(log^2 n), O(n log n)> solution

Sum(r1=1, c1=5, r2=2, c2=7)

2. Query $\boxed{1} + \boxed{2}$

$\boxed{1} + \boxed{2} \longrightarrow$

6	6	2	6	-6	-2	8	8

Query(7) = 8 + -2 + 6 = 12
Query(5-1) = Query(4) = 6

Sum = Query(7) - Query(4) = 12 - 6 = 6

A slower <O(log^2 n), O(n log n)> solution

Sum(r1=1, c1=5, r2=2, c2=7)

2. Query $\boxed{1}$ + $\boxed{2}$

$\boxed{1}$ + $\boxed{2}$ ⟶

6	6	2	$\boxed{6}$	-6	-2	8	8

Query(7) = 8 + -2 + 6 = 12
Query(5-1) = Query(4) = 6

Sum = Query(7) - Query(4) = 12 - 6 = 6

Check your answer :)

c=1							8
3	0	1	-1	-3	2	4	-2
3	0	1	-1	-3	2	4	-2
3	0	1	-1	-3	2	4	-2

Sum(r1=1, c1=5, r2=2,c2=7)
= -3 + 2 + 4 + -3 + 2 + 4
= 6

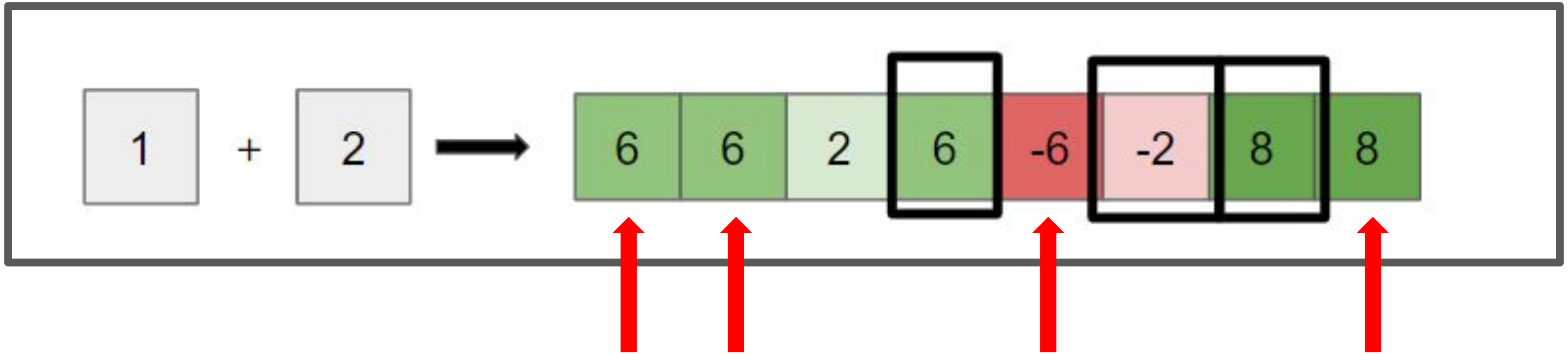
A slower <O(log^2 n), O(n log n)> solution

```
FenwickTree<int> tree[n+5];

FenwickTree<int> tree_query(int r) {
    FenwickTree<int> sum;
    for (int i = r; i > 0; i -= i & (-i)) {
        sum = sum + tree[i];
    }
    return sum;
}

int orthogonal_range_sum(int r1, int c1, int r2, int c2) {
    return (tree_query(r2) - tree_query(r1 - 1)).sum(c1, c2);
}

void update(int r, int c, int v) {
    for (int i = r; i <= n; i += i & (-i)) {
        tree[i].update(i, v);
    }
}
```

A <O(log^2 n), O(log^2 n)> solution

Reminder: **Most** of the cells in the Fenwick Tree is **untouched**!

- Idea: Sum the O(log n) cells you care about



Don’t have to calculate these!!

A <O(log^2 n), O(log^2 n)> solution

```
FenwickTree<int> tree[n+5];

int tree_query(int r, int c1, int c2) {
    int sum = 0;
    for (int i = r; i > 0; i -= i & (-i)) {
        sum = sum += tree[i].sum(c1, c2);
    }
    return sum;
}

int orthogonal_range_sum(int r1, int c1, int r2, int c2) {
    return tree_query(r2, c1, c2) - tree_query(r1 - 1, c1, c2);
}

void update(int r, int c, int v) {
    for (int i = r; i <= n; i += i & (-i)) {
        tree[i].update(i, v);
    }
}
```

A more compact solution

This is **not** 2d Fenwick Tree you typically find online.

What you typically find:

- **Query** returns sum in the rectangle bounded by **(1, 1)** and **(r, c)**
- This follows the exact same principle we discussed
    - Exact code left as practice for the readers

```
struct TwoDimFenwickTree {
    int update(int r, int c, int v);
    int query(int r, int c);
}
```

A more compact solution

Sum can be found using **Principle of Inclusion & Exclusion** (PIE)

- Same technique from 2d Partial Sum
- You should be **very very familiar** with this as well

```
struct TwoDimFenwickTree {
    int update(int r, int c, int v);
    int query(int r, int c);
    int sum(int r1, int c1, int r2, int c2) {
        return query(r2, c2) - query(r1-1, c2) - query(r2, c1-1) + query(r1, c1);
    }
}
```

Generalization

Recipe for designing DS for higher dimensional data

1. Find recurring query patterns
2. Nest data structures
3. Remove unnecessary access

Generalization

A test for you: **Derive** and **argument** why each of them is useful

1. 3d Fenwick Tree?
2. 2d Segment Tree?
3. Segment tree of sets?
4. 2d Binary Search Trees?

Some Tips for Higher Dimensional DS task

1. Most problems don't give you the entire grid
- Often gives a set of *q* operations on a large imaginary grid (10^5 * 10^5)
- You **don't** use **most** of the tree nodes

E.g.

Orthogonal Range query on a 10^5 * 10^5 grid of 0 s

- **Q** operations of **update** / **sum**

You only work on Q * log^2(10^5) cells

Some Tips for DS tasks

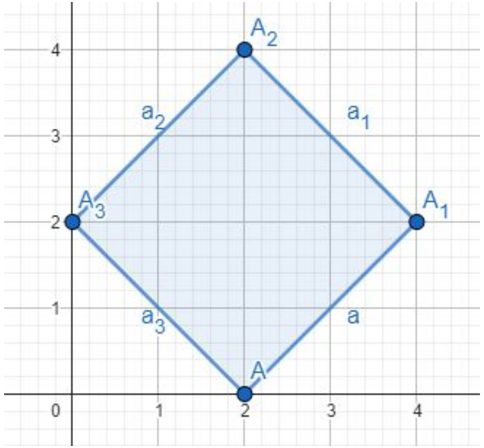
1. Most problems don’t give you the entire grid

- Often gives a set of *q* operations on a large imaginary grid (10^5 * 10^5)
- You **don’t** use **most** of the tree nodes

- **Dynamically** allocate tree node only when **you need one**!
- Use coordinate compression

Some Tips for DS tasks

2. Orthogonal Range Query is an absolute beast!
   - Try transforming data format into orthogonal range queries!
   - Querying this rhombus? → Try rotating by 45°
   - Now solve M1952!



Some Tips for DS tasks

3. Segment Tree is (probably) all you need!
- Don’t be limited to **addition!**
- Segment Trees support any **associative** operations
  - **Associative**: x * (y * z) = (x * y) * z

- Try T192, M1839

- You should know this very well now but it only gets more fun in higher dimension!

Some Tips for DS tasks

4. Don’t be limited to Geometry!

   Higher dimensional DS can be very useful for:

- Obscurely formatted tasks
- DP optimizations
- Basically anywhere that seems to involve more than 1 index

- Try AP181

Break Time!

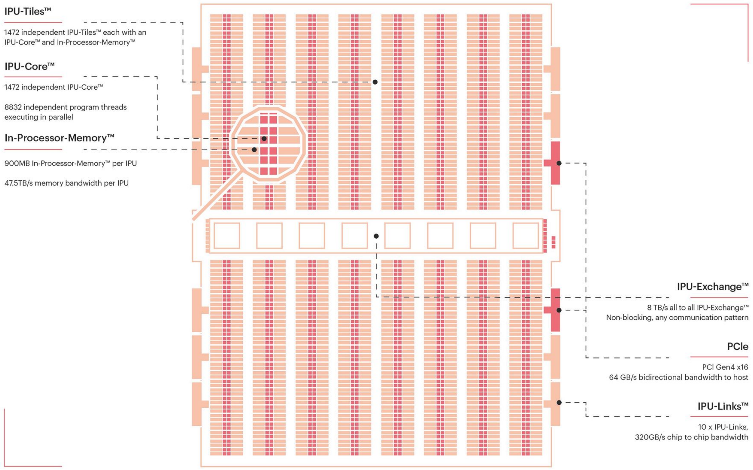
- **10** mins break :)

Into the realm of parallel algorithm

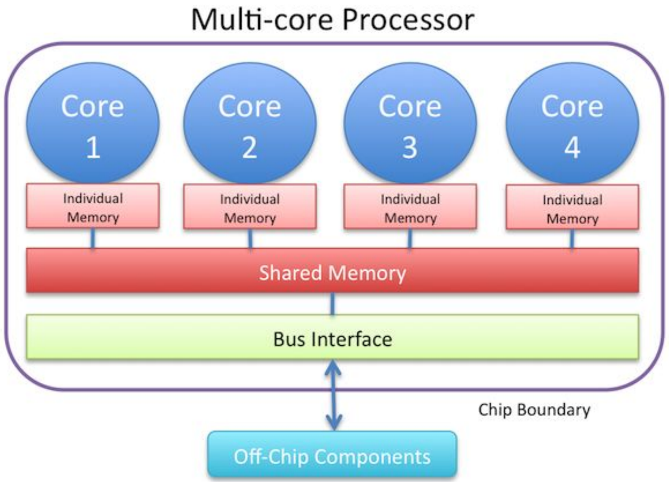
- We have mostly algorithms running on a single machine
- Modern machines have multiple (4? 8?) cores
- Wasting a lot of computation power :(

Modern Computer Architecture

Graphcore IPU (1472 cores)



Consumer Multicore CPU (4 cores)



Some easily parallelizable programs

- Parallel Min-Finding

```
def find_min(a: list):
    n = len(a) // 2

    spawn thread1 running find_min(a[n...])

    result = find_min(a[...n])

    wait for thread1 to terminate

    result = min(result, thread1.result)

    return result
```

- Assuming you have infinite core, this runs in logarithmic time :)

Some hard-to-parallelize programs

Applying function *f* to an integer *a* by *n* times

```
int apply_f(f: int -> int, a: int, n: int) {
    for (int i = 1; i <= n; ++i) {
        a = f(a);
    }
    return a;
}
```

apply_f(fun x -> x + 1, 2, 3) = (((2+1)+1)+1) = 5

- Data is dependent on previous computation -> hard to parallelize

Designing easily parallelizable programs

Some thoughts

- Data sharing / dependeny -> bad : (

One very simple strategy:

- Observe **common pattern**: Given some batch of data, compute statistics
- Rearrange and split data into segments that are **easy to compute**
- The result of the function calls are **easy to combine**
- See Min-Finding

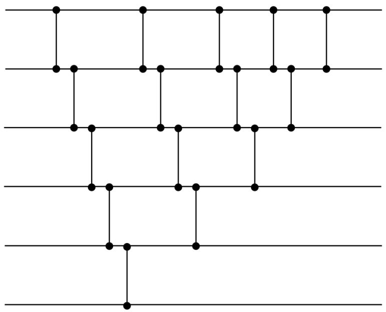
Designing easily parallelizable programs

We can easily refactor programs to be friendlier for parallelization

Sorting Network:

- **Horizontal Line**: Input
- **Vertical Lines**: Comparison

- Hard to split computation for bubble sort

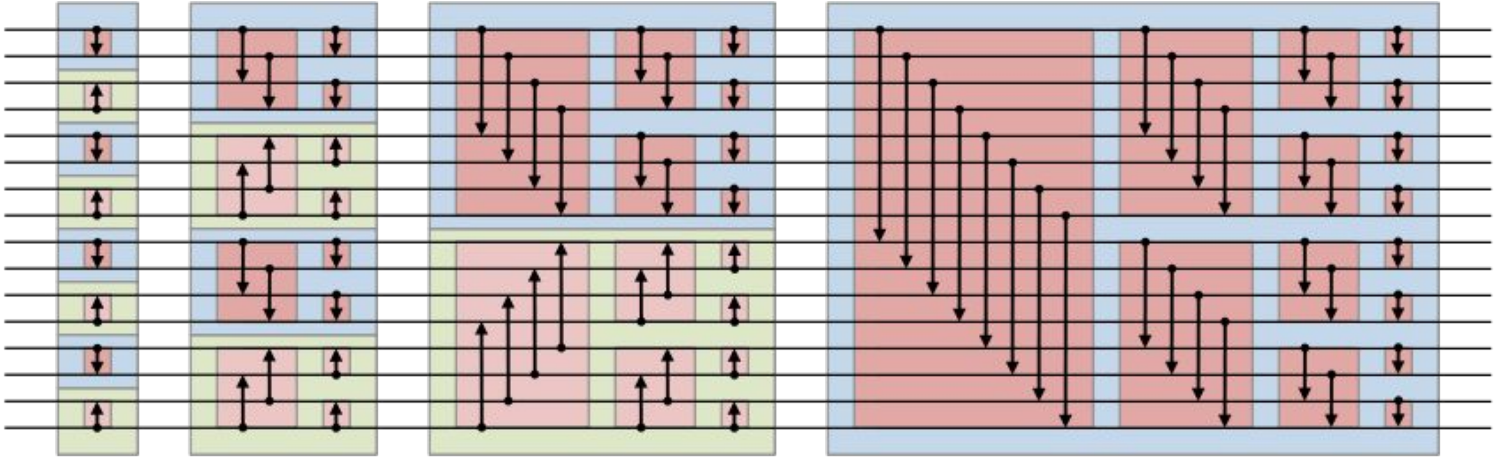


Sorting network for bubble sort

Designing easily parallelizable programs

Bitonic sorter

- By rewiring the network carefully, you can go very far
- Easy to parallelize, and sometime easier implementation
- Similar techniques in IOI’21 bit-shift register (interactive)



Advanced Divide-and-Conquer

A recurring theme in competitive programming:

- Given a batch of data / operations
- Calculate the result of each operation
- Divide-and-conquer unlocks new classes of **batch / offline processing** techniques

CDQ Divide-and-Conquer

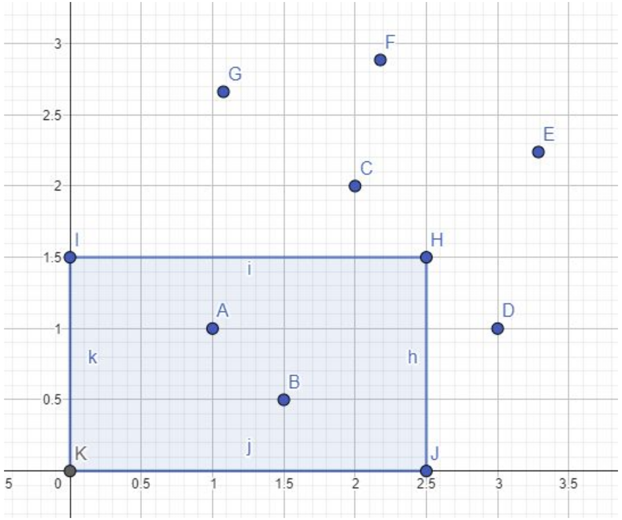
Example:

Given an array of 2d pairs a[n] = {(x1, y1), (x2, y2), (x3, y3), ... }

For each 1 <= i <= n:

- Find the number of j < i such that $x_j <= x_i$ *and* $y_j <= y_i$
- For simplicity write (x1, y1) <= (x2, y2) *if x1 <= x2 and y1 <= y2*

CDQ Divide-and-Conquer



Thinking the problem geometrically

Oh this is easy, use a 2d segment tree

To make your life harder:

- Make it 3d
- Put some max/min in the operations so that 3d fenwick doesn't work
- **Takeaway**:

  Just a **toy** problem for teaching

CDQ Divide-and-Conquer

Rearranging data…

1. Let's first sort the elements in **ascending** order of **x**

- (2, 3), (1, 1), (5, 4), (6, 3), (8, 1), (7, 5), (5, 8), (3, 7)
- (1, 1), (2, 3), (3, 7), (5, 4), (5, 8), (6, 3), (7, 5), (8, 1)

CDQ Divide-and-Conquer

Splitting the data…

- left = (1, 1), (2, 3), (3, 7), (5, 4)
- right = (5, 8), (6, 3), (7, 5), (8, 1)

Now assume we are able to calculate the **contribution** within the **segment**

i.e. for (7, 5), we identify (6, 3) <= (7, 5) but no other

CDQ Divide-and-Conquer

All that remains is combining results...

- left = (1, 1), (2, 3), (3, 7), (5, 4)
- right = (5, 8), (6, 3), (7, 5), (8, 1)

Observation:

1. The points in the **left** segment contribute to the **right**
2. The points in the **right** segment does **not** contribute to the **left**
3. All points in **left** segment has x <= those on the **right**

CDQ Divide-and-Conquer

Combining results…

We can reduce the problem to:

Given a fixed set, ***S***, of **y-coordinate** (in the **left** segment)

For each element **i** on the **right** segment:

- Find #element (**e**) in **S** such that **e <= a[i].y**

Why this works: We don't have to worry about the **x-coordinate** anymore

CDQ Divide-and-Conquer

- left = (1, 1), (2, 3), (3, 7), (5, 4) right = (5, 8), (6, 3), (7, 5), (8, 1)
- S = {1, 3, 7, 4}

CDQ Divide-and-Conquer

- left = (1, 1), (2, 3), (3, 7), (5, 4)

  right = **(5, 8)**, (6, 3), (7, 5), (8, 1)

- S = {1, 3, 7, 4}

- For point **(5, 8)** on the **right**
  - **4** elements in **S** are smaller than **8**
  - Corresponding to: (1, 1), (2, 3), (3, 7), (5, 4)

CDQ Divide-and-Conquer

- left = (1, 1), (2, 3), (3, 7), (5, 4)

right = (5, 8), **(6, 3)**, (7, 5), (8, 1)

- S = {1, 3, 7, 4}

- For point **(6, 3)** on the **right**
  - **2** elements in **S** are smaller than **3**
  - Corresponding to: (1, 1), (2, 3)

CDQ Divide-and-Conquer

- left = (1, 1), (2, 3), (3, 7), (5, 4)

  right = (5, 8), **(6, 3)**, (7, 5), (8, 1)

- S = {1, 3, 7, 4}

- How to compute the query on **S**?

- 1d Segment Tree / Fenwick Tree

CDQ Divide-and-Conquer

Back to our **assumptions...**

- left = (1, 1), (2, 3), (3, 7), (5, 4)
- right = (5, 8), (6, 3), (7, 5), (8, 1)

Now assume we are able to calculate the **contribution** within the **segment**

CDQ Divide-and-Conquer

Back to our **assumptions...**

- left = (1, 1), (2, 3), (3, 7), (5, 4)
- right = (5, 8), (6, 3), (7, 5), (8, 1)

Now assume we are able to calculate the **contribution** within the **segment**

- We don't quite have to **care** about this

CDQ Divide-and-Conquer

Claim: As long as the combiner is correct, the entire algorithm is correct

Why?

- By Induction :)

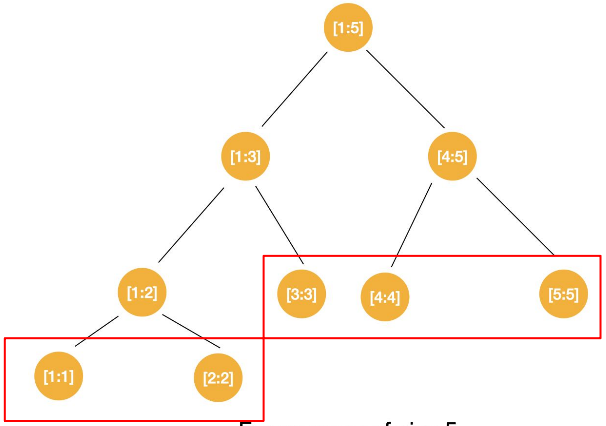
CDQ Divide-and-Conquer

Claim: As long as the combiner is correct, the entire algorithm is correct

Why?

- By Induction :)

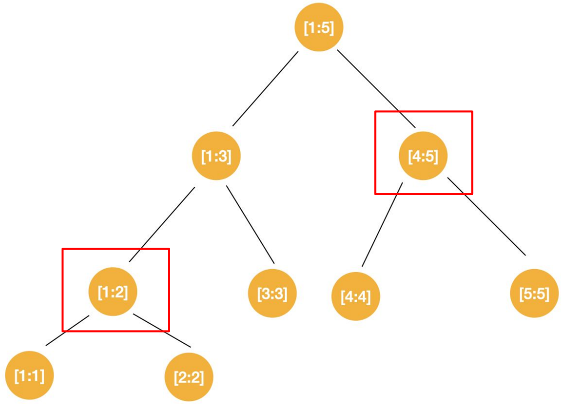
- Leaf node:
- 0 contribution within segment



For an array of size 5

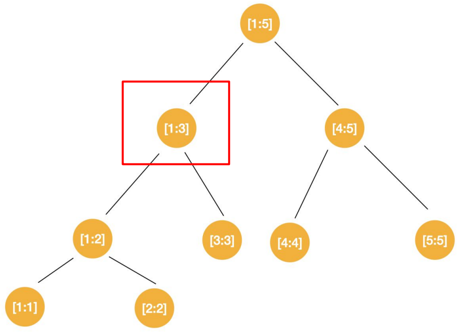
CDQ Divide-and-Conquer

- Leaf node + 1:
- All contributions within children is
- All contributions across children is



For an array of size 5

CDQ Divide-and-Conquer



For an array of size 5

- Leaf node + 2:
- All contributions within children is counted
- All contributions across children is accounted for (by combiner)
- **Ok you should see where this is going**

CDQ Divide-and-Conquer

- Useful techniques for dimension reduction via rearranging + D&C

- 3 steps:
  1. Rearrange
  2. Split
  3. Combine

CDQ Divide-and-Conquer

We haven’t covered any **offline processing** techniques yet?

CDQ Divide-and-Conquer

APIO 2019 – Street Lamps (modified)

Given an **n**-bit array initialized with all 0's i.e. *a = [0, 0, ...]*

For each of the next **Q** time unit, perform **one** operation (aka perform **Q** operations)

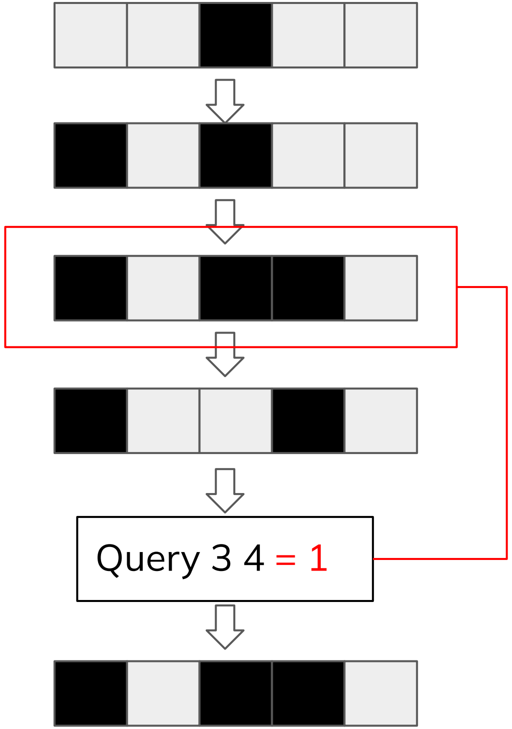
1. toggle i → a[i] = a[i] xor 1

2. query l r →
   Count the #**unit** of **time** before the query where in **a[l]=a[l+1]=...=a[r]=1**
   (i.e. all elements between l and r are all 1's)

APIO 2019 – Street Lamps (modified)

Example:

n = 5
toggle 3
toggle 1
toggle 4
toggle 3
query 3 4
toggle 3



APIO 2019 – Street Lamps (modified)

Time is just another dimension you can manipulate

General hints:

- Try to rewrite out relations as tuple ordering, e.g.

  $\boxed{(x1, y1) <= (x2, y2) \textit{ if } x1 <= x2 \textit{ and } y1 <= y2}$

- Think Geometrically! Draw diagrams!
- We'll come back later if time permits

APIO 2019 – Street Lamps (modified)

Time is just another dimension you can manipulate

More specific hints:

- What can you arrange?
- Is life easier if all **toggles** happen **before queries**
- You will need **BOTH** CDQ and 2d Fenwick Trees :)

Q&A + Break Time!

- **20** mins break :)
- Try working on Street Lamps

Persistent Data Structures

Playing with time...

Given an array of **n** elements a[n] = {a1, a2, ..., an}
Perform **Q** operations:

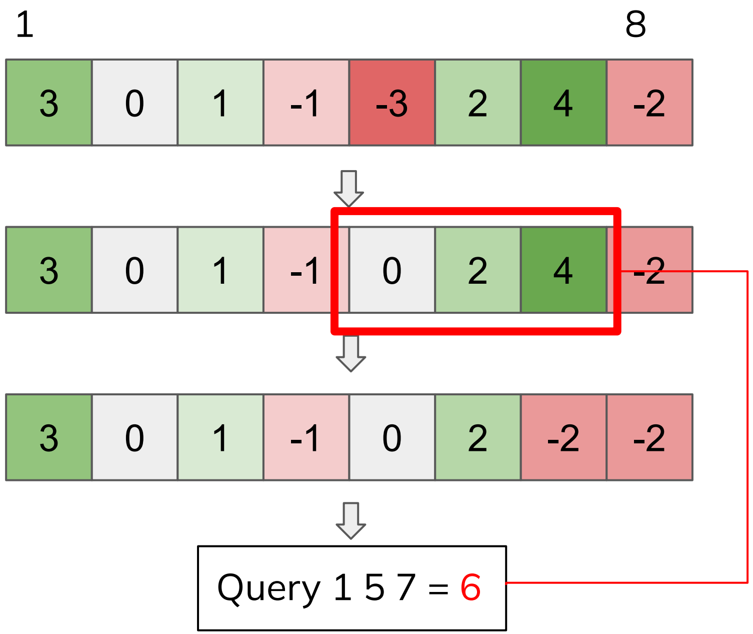
1. Update i, v → set a[i] = v
2. Query t l r → For the array **a** after **t** operations, find the sum from **l** to **r**?

Persistent Data Structures

Example:

a = {3, 0, 1, -1, -3, 2, 4, -2}

update 5 0
update 7 -2
query 1 5 7



Persistent Data Structures

This would be **very easy without** the time element

Can we still view **old** copies of the data structure **after update?**

Persistent Data Structures

This would be **very easy without** the time element

- Just use a simple 1d segment tree

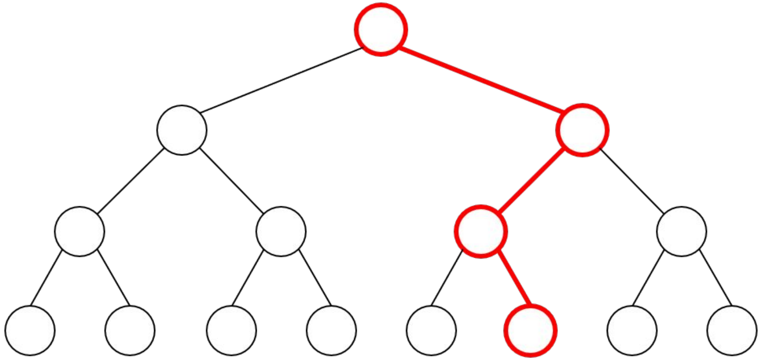
Can we still view **old** copies of the data structure **after update?**

Persistent Data Structures

Observe that only a small amount of nodes change per update

On the creation of a new “**version**“

- Create new node **only** for **modified ones**

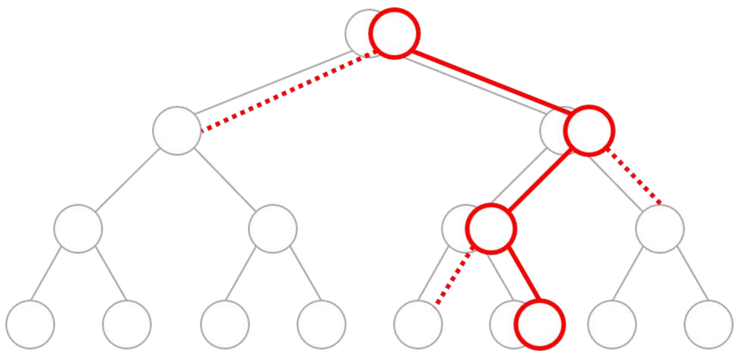


Persistent Data Structures

Observe that only a small amount of nodes change per update

On the creation of a new "**version**"

- Create new node **only** for **modified ones**

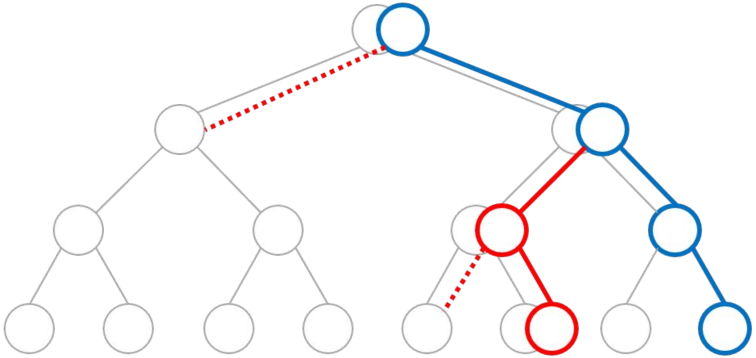


Persistent Data Structures

Observe that only a small amount of nodes change per update

On the creation of a new “**version**“

- Create new node **only** for **modified ones**

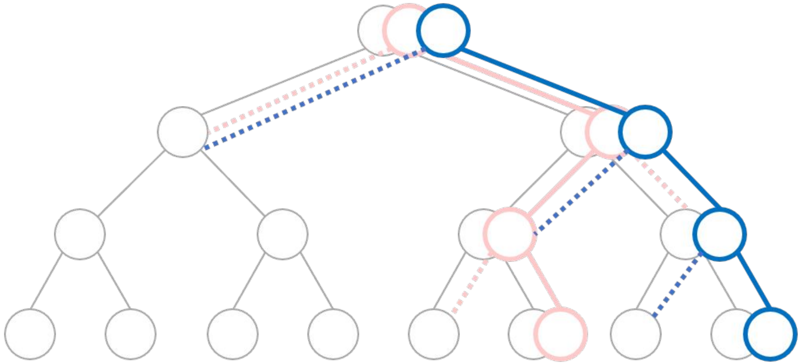


Persistent Data Structures

Observe that only a small amount of nodes change per update

On the creation of a new “**version**“

- Create new node **only** for **modified ones**



Different level of persistence

Partially Persistence:

- **Modify** newest version, **access** old version

Fully Persistence:

- **Modify** newest version, **branch from** old version
- A “tree” of timeline :)
- Persistent Segment Tree falls here

Different level of persistence

Confluently Persistence:

- Fully Persistence
- Can I merge two timelines?

Retroactive Data Structure:

- Oh I made a mistake in a past update
- => Modify past operations
- Changes reflect in the newest version!
- **Real** time travel?

Persistence Practice Problem

M1842 – Another RMQ

APIO 2019 – Land of Rainbow Gold

NOI 2018 – 归程