Data Structure(III)

Ian

Things that we would talk about

- **Sparse Table**
- **Segment tree**
- **Binary indexed tree**

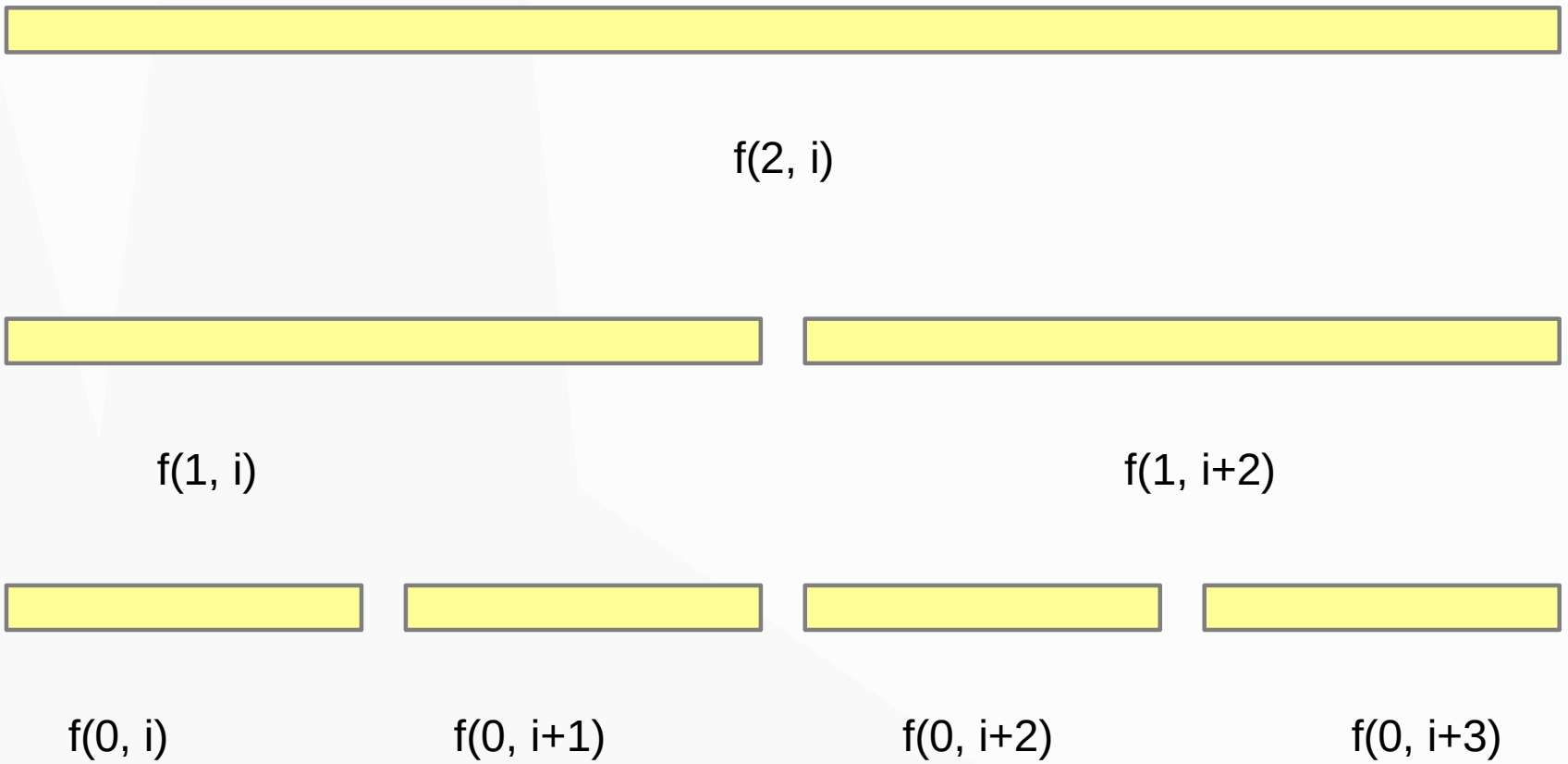
Sparse Table

- **Sparse Table is a data structure that support query but not update**
- **It require O(N log N) to precompute**
- **But can answer query in O(1) !!!**
- **Very good when u need to perform many query**

Sparse Table

- Assume we now want to solve the range minimum query problem
- Denote f(k, i) as the minimum element from i to i+2^k –1
- f(0, i) is obvious, as it ask for minimum in [i, i] which equals to a[i].
- For f(1, i), f(1, i) = min(f(0, i), f(0, i+1))
- Generally, we can combine information of f(k, i) and f(k, i+2^k) to f(k + 1, l)
- So we now can use O(N log N) to precompute f(k, i)

Sparse Table



Sparse Table

- **Now for the query part:**
- **Suppose we do query(l, r)**
- **Obviously, if (r – l + 1) is 2^k, f(k, l) will be the answer**
- **If (r – l + 1) couldn’t be represented by 2^k:**
  - **Let k be the max. integer such that 2^k <= (r – l + 1)**
  - **min(f(k, l), f(k, r – 2^k + 1)) will be the answer**
  - **Just combining information of two parts**
  - **Note that two interval might overlap so it only work when function f is associative**
  - **Otherwise you need to do O(log n) query (splitting the interval to log n sub-interval)**

Sparse Table for LCA

- For the lowest common ancestor problem, we could use sparse table to solve it
- f(k, v) = u where dist(u, v) = 2^k and u is v's ancestsor
- f(0, v) = parent of v. f(i + 1, v) = f(i, f(i, v)). The 2^i parent of the 2^i parent for node v is its 2^(i + 1) parent

Sparse Table for LCA

- For finding lca of u and v. Assume depth(u) <
- depth(v) Keep lifting v using f(k, v) until depth(u) =
- depth(v)
- v = u then u is the answer

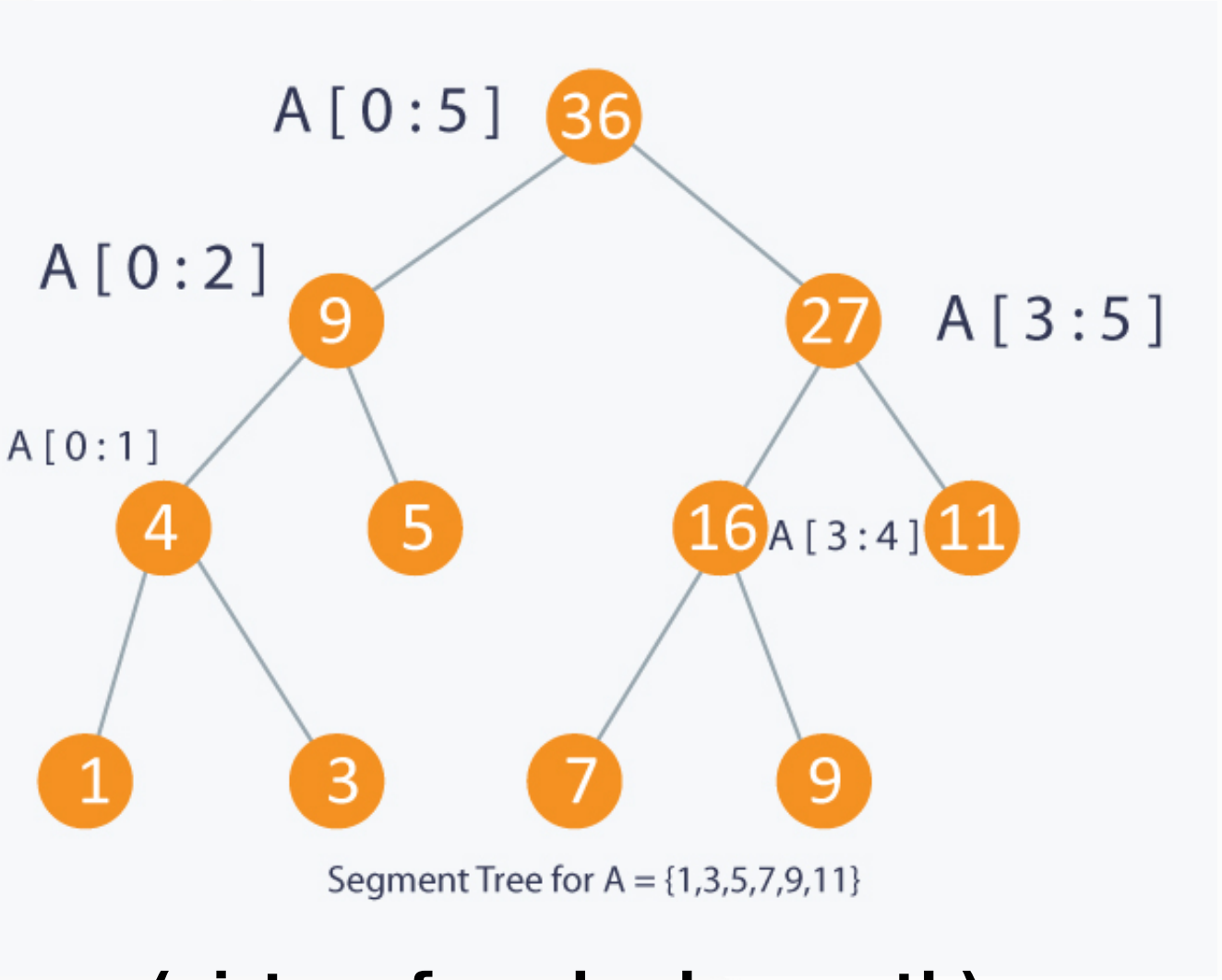
  Otherwise, iterate k from $\log_2(n)$ to 0, if f(k, v) <>
- f(k, u), lift v and u to f(k, v) and f(k, u)

  f(0, v) would be the answer

Segment Tree

- **Segment tree is a data structure that support range query and update**
- **It is a binary tree**
- **Each node represent a segment.**
- **Assume node v maintain data of [l, r]**
  - **Left child maintain [l, mid]**
  - **Right child maintain [mid + 1, r]**
  - **Where mid = (l + r) / 2**
- **As each time interval is divided by 2, height of tree is log N**

Segment Tree



(picture from hacker earth)

Classical Problem for Segment Tree

- **Range minimum(maximum) query**
  - **Given an integer A**
  - **Query(l, r)**
    - **Ask for minimum element in [l; r]**
  - **Update(id, val)**
    - **Update element in position id to val**
- **EX : M0921**

Classical Problem for Segment Tree

- **Maximum subarray problem**
  - **Given an integer array A**
  - **Query(l, r)**
    - **Find maximum sum of subarray [a; b] such that l <= a <= b <= r**
  - **update(id, val)**
    - **Update element in position id to val**
- **EX : M0923**

Classical Problem for Segment Tree

- **Sweep Line**
- **Given N rectangle, find union area**
- **http://codeforces.com/blog/entry/20377**
- **EX : M1633**

Implementation

- **update(id, x, y, pos, val) // update value in pos to val**

```
node id maintain [x; y]
If x == y
    a[pos] = val
    return
mid =  (l + r) / 2
If (pos <= mid) update(id * 2, x, mid, pos, val)
else update(id * 2 + 1, mid + 1, y, pos, val)
a[id] = max(a[id * 2], a[id * 2 + 1])
```

Implementation

- **query(int id, int x, int y, int l, int r) // find ans in [l; r]**

```
if (out of range)
    return -1
if (l <= x and y <= r)
    return a[id]
mid = (l + r) / 2
return
max(query(id * 2, x, mid, l, r), query(id * 2 + 1, mid + 1, y, l, r)
```

Practice Problems for Segment Tree

- **CF 339D**
- **CF 380C**

Findind LCA using Segment Tree

- https://www.topcoder.com/community/data-science/data-science-tutorials/range-minimum-query-and-lowest-common-ancestor/
- http://codeforces.com/blog/entry/16221

Lazy Propagation

- **In previous slide, segment tree could only deal with problem that is:**
  - **Range query and point update. Or**
  - **Point query and range update.**
- **To solve problems with range query and range update, we would need to use this “lazy propagation technique”**

Lazy Propagation

- **The name of the technique just said it all**
- **We add a “lazy” variable into each node in segment tree**
- **The main idea is that, we only do update when it is needed (lazy)**
- **If current interval has been updated and haven’t apply this change to its sub-interval, then apply. Otherwise we don’t care.**

Implementation

push(id)

```
if (lazy[id] == -1) // no change
    return
lazy[left child] = lazy[right child] = lazy[id]
a[left child] = a[right child] = lazy[id]
lazy[id] = -1
```

Implementation

- **update(id, x, y, l, r, val) // update value in [l, r] to val**

```
node id maintain [x; y]
If [x, y] fully lies in [l, r]
    a[id] = val
    lazy[id] = val
    return
mid =  (l + r) / 2
push(id)
update(id * 2, x, mid, l, r, val)
update(id * 2 + 1, mid + 1, y, l, r, val)
a[id] = max(a[id * 2], a[id * 2 + 1])
```

Implementation

- **query(int id, int x, int y, int l, int r) // find ans in [l; r]**

```
if (out of range)
    return -1
if (l <= x and y <= r)
    return a[id]
mid = (l + r) / 2
push(id)
return
max(query(id * 2, x, mid, l, r), query(id * 2 + 1, mid + 1, y, l, r)
```

Lazy Propagation

- **More info:**
  - https://www.geeksforgeeks.org/lazy-propagation-in-segment-tree/

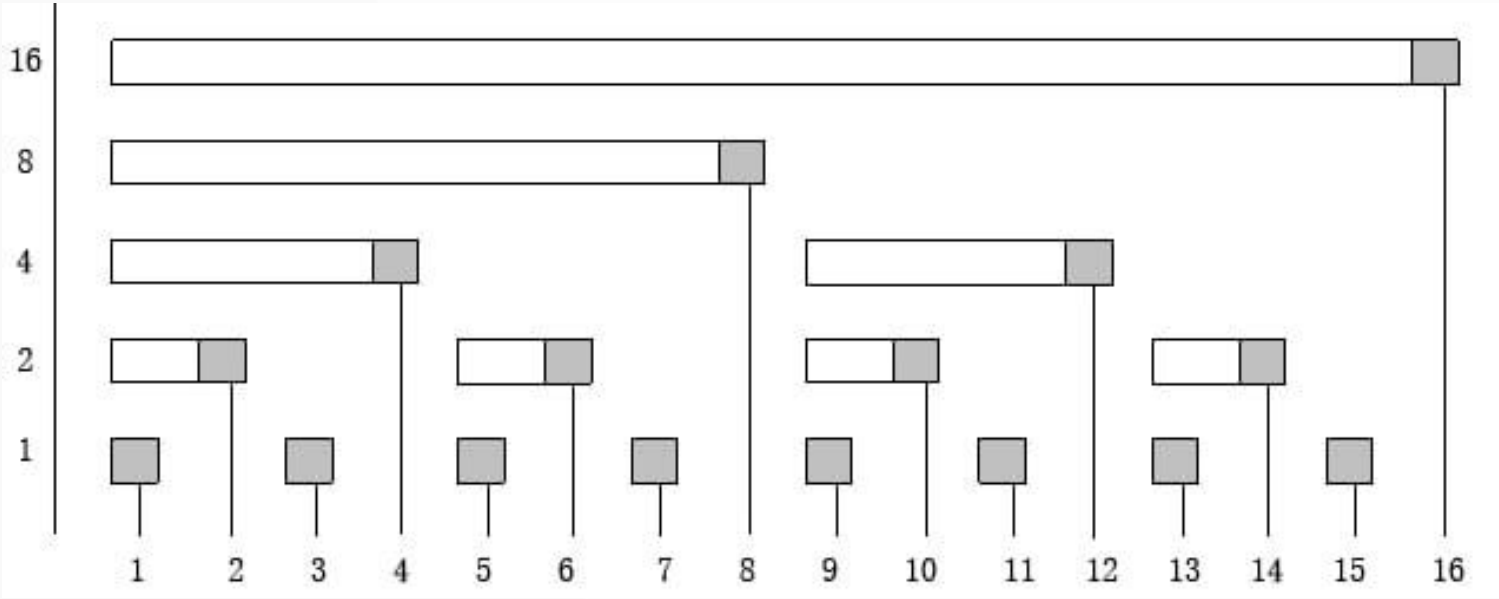
Binary Indexed Tree

- **It is a simplified segment tree**
- **Define lowbit(x) as the value of the rightmost bit in binary representation of x**
- **Let x = 22 = 10110_2, lowbit(x) = 00010_2 = 2**
- **Node x maintain information for [x – lowbit(x) + 1, x]**
- **lowbit(x) = x & -x**

Binary Indexed Tree

- **Given an array A**
- **Support two operation**
  - **update(id, val)**
    - **Add val to A[id]**
  - **sum(id)**
    - **Find sum of A from 1 to id**

BIT Visualization



Picture from https://www.hrwhisper.me/binary-indexed-tree-fenwick-tree/

Implementing BIT

- **Add (id, val)**

  while id <= N

  add val to BIT[id]

  id = id + id & -id ←--- adding its lowbit

- **Sum (id)**

  ans = 0

  while id > 0

  add BIT[id] to ans

  id = id – id & -id ←--- subtracting its lowbit

Binary Indexed Tree

- https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/

BIT Practice Problems

- **CF 830B**

2D Data Structure

- **For segment tree and BIT, you could actually extend them to 2D and solve 2D problems**
- **Using BIT as an example. If we extend it to two dimension, it just like a 1D BIT but every element would be a BIT instead of an array cell.**

Practice Problem

- **HKOJ I0111**