Data Structures I

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Sydney

Term 3, 2023

Table of Contents

How does <vector> work?

- Vectors are dynamic arrays
- Random access is $O(1)$, like arrays
- A vector is stored *contiguously* in a single block of memory
- Supports an extra operation `push_back()`, which adds an element to the end of the array
- STL implements a templated vector in `<vector>`

How does `push_back()` allocate memory?

- Do we have enough space allocated to store this new element? If so, we're done: $O(1)$.
- Otherwise, we need to allocate a new block of memory that is big enough to fit the new vector, and copy all of the existing elements to it.
- This is an $O(n)$ operation when the vector has n elements. How can we improve?
- If we double the size of the vector each reallocation, we perform $O(n)$ work once, and then $O(1)$ work for the next $n - 1$ operations, an average of $O(1)$ per operation.
- We call this time complexity *amortised* $O(1)$.

Aside: amortised complexity

- How is amortised complexity different from average case complexity?
- For an expected constant time operation (e.g. hash table lookup), it may still be possible for n consecutive operations to each take $O(n)$ time, for a total time of $O(n^2)$.
- This is not possible with amortised complexity. An individual operation might take $O(n)$ time, but n consecutive operations are *guaranteed* to take $O(n)$ time in total.

Vectors: Usage


```
#include <cassert>
#include <vector>
using namespace std;

int main() {
    vector<int> v;
    for (int i = 0; i < 10; i++) v.push_back(i*2);
    v[4] += 20;
    assert(v[4] == 28);
}
```

Table of Contents

Stacks

- Supports `push()` and `pop()` operations in $O(1)$
- LIFO (last in, first out)
- STL implements a templated stack in `<stack>`
- Equivalently, you can use an array or vector to mimic a stack, with the advantage of allowing random access

Queues

- Supports `push()` and `pop()` operations in $O(1)$
- FIFO (first in, first out)
- STL implements a templated queue in `<queue>`
- Equivalently, you can use an array or vector to mimic a queue, with the advantage of allowing random access

Stacks and Queues: Usage


```
#include <cassert>
#include <queue>
#include <stack>
using namespace std;

int main() {
    stack<int> stk;
    queue<long long> que;

    stk.push(1);
    stk.push(2);
    assert(stk.top() == 2);
    stk.pop();
    assert(stk.top() == 1);
    assert(stk.size() == 1);
    assert(!stk.empty());

    que.push(1);
    que.push(2);
    assert(que.front() == 1);
}
```

Table of Contents

Sets

- STL's `<set>` is a set with $O(\log n)$ random access
- Internally implemented as a red/black tree of set elements
- Unfortunately doesn't give you easy access to the underlying tree - iterator traverses it by infix order
- C++11 adds `<unordered_set>`, which uses hashing for $O(1)$ average case ($O(n)$ worst case) random access
- Main advantage of `<set>` is it keeps the data ordered, hence has `lower_bound(x)` and `upper_bound(x)` which returns the next element not less than (resp. greater than) x
- `<multiset>` and (C++11) `<unordered_multiset>` are also available

Maps

- STL's `<map>` is a dictionary with $O(\log n)$ random access
- Internally implemented with a red/black tree of (key,value) pairs
- Unfortunately doesn't give you access to the underlying tree - iterator traverses it by infix order
- C++11 adds `<unordered_map>`, which uses hashing for $O(1)$ average case ($O(n)$ worst case) random access
- Main advantage of `<map>` is it keeps the data ordered, hence has `lower_bound(x)` and `upper_bound(x)` which returns the next element whose key is not less than (resp. greater than) x
- `<multimap>` and (C++11) `<unordered_multimap>` are also available

Sets and Maps: Usage


```
#include <iostream>
#include <map>
#include <set>
using namespace std;

set<int> s;
map<int, char> m;

int main() {
    s.insert(2); s.insert(4); s.insert(1);
    m = {{1,'a'}, {4,'c'}, {2,'b'}};
    // Check membership:
    cout << (s.find(2) != s.end()) << ' ' << (s.find(3) != s.end()) << '\n'; //
        1 0
    // NOT binary_search(s.begin(), s.end(), 2), which takes linear time

    // Access map:
    cout << m[1] << '\n'; // 'a'
    // WARNING: Access to non-existent data just silently adds it, avoid this.
    // cout << m[3] << '\n'; // null character

    // Lower and upper bounds:
    cout << *s.lower_bound(2) << '\n'; // 2
    // NOT *lower_bound(s.begin(), s.end(), 2), which takes linear time
    cout << *s.upper_bound(2) << '\n'; // 4
    auto it = m.lower_bound(2);
    cout << it->first << ' ' << it->second << '\n'; // 2 b

    // Move around with prev/next or increment/decrement
    cout << prev(it)->first << '\n'; // 1
    cout << (++it)->first << '\n'; // 4
}
```

- One of the main problems with set and map is they don't track index information.
- So you can't query what the k-th number is or how many numbers are $< x$.
- Most SBBSTs can be modified to track this metadata. But we do not want to implement a SBBST.

Order Statistic Trees

- There is a fix in GNU C++. So it is not a C++ standard but pretty widespread.
- Contained in an extension called "Policy Based Data Structures".
- In headers:

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
```

- Details are pretty technical, fortunately we don't need to know them.

New data structure:

```
typedef tree<int, null_type, less<int>, rb_tree_tag,
             tree_order_statistics_node_update>
             ordered_set;
```

- Key type: `int`
- No mapped type (a set not a map)
- Comparison: `less<int>`
- `rb_tree_tag`: Implemented as a red-black tree, guarantees O(log n) performances
- `tree_order_statistics_node_update`. The magic: tells it to update order statistics as it goes.

Order Statistic Trees

Essentially a set/map with 2 extra operations:

- `find_by_order(x)`: Find the x-th element, 0-indexed.
- `order_of_key(x)`: Output the number of elements that are $< x$.
- Both are $O(\log n)$ still!
- Furthermore, in other regards they still behave like a set/map!

Order Statistic Trees: Usage


```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
using namespace std;

typedef tree<int, null_type, less<int>, rb_tree_tag,
            tree_order_statistics_node_update> ordered_set;

ordered_set myset;
int main() {
    myset.insert(2);
    myset.insert(4);
    myset.insert(1);
    printf("%d\n", *(myset.find_by_order(0))); // 1
    printf("%d\n", (int)myset.order_of_key(3)); // 2
    printf("%d\n", (int)myset.order_of_key(4)); // 2
    printf("%d\n", (int)myset.size()); // 3
}
```

Data Structures I

Vectors

Stacks and Queues

Sets and Maps

Heaps

Basic Examples

Example Problems

Union-Find

Range Queries and Updates


```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
using namespace std;

typedef tree<int, char, less<int>, rb_tree_tag,
             tree_order_statistics_node_update> ordered_map;

ordered_map mymap;
int main() {
    mymap[2] = 'a';
    mymap[4] = 'b';
    mymap[1] = 'c';
    pair<int, char> pic = *mymap.find_by_order(0);
    printf("%d %c\n", pic.first, pic.second); // 1 c
    printf("%d\n", (int)mymap.order_of_key(3)); // 2
    printf("%d\n", (int)mymap.order_of_key(4)); // 2
    printf("%d\n", (int)mymap.size()); // 3
}
```

Table of Contents

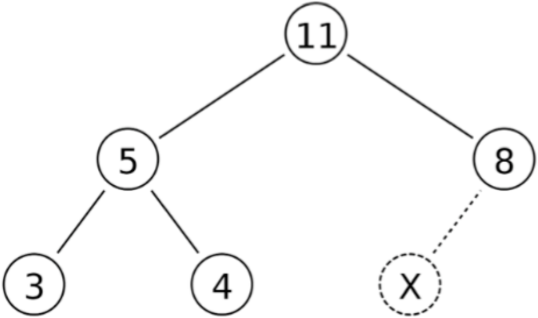
- Supports `push()` and `pop()` operations in $O(\log n)$, `top()` in $O(1)$.
- `top()` returns the value with highest priority
- Is usually used to implement a priority queue data structure
- STL implements a templated priority queue in `<queue>`
- The default is a max heap - often we want a min heap, so we declare it as follows:

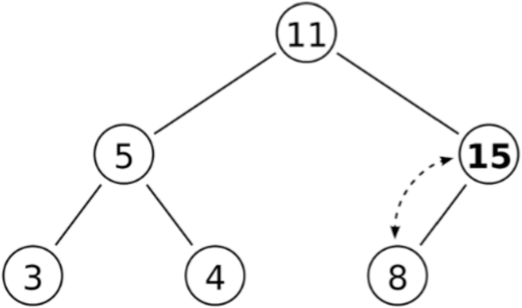
```
#include <queue>
priority_queue <T, vector<T>, greater<T>> pq;
```

- It's significantly more code to write a heap yourself, as compared to writing a stack or a queue, so it's usually not worthwhile to implement it yourself

- The type of heaps usually used is more accurately called a *binary array heap* which is a binary heap stored in an array.
- It is a binary tree with two important properties:
  - **Heap property:** the value stored in every node is greater than the values in its children
  - **Shape property:** the tree is as close in shape to a complete binary tree as possible

- Operation implementation
  - `push(v)`: add a new node with the value *v* in the first available position in the tree. Then, while the heap property is violated, swap with parent until it's valid again.
  - `pop()`: the same idea (left as an exercise)


11
5
8
3
4
X


11
5
15
3
4
8

Heaps

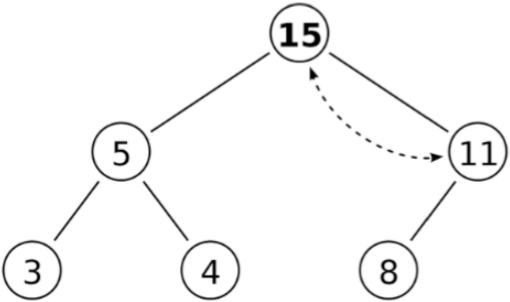


Table of Contents

Basic Uses

- Most uses fall out naturally from the use case.
- Vectors: Use everywhere.
- Stacks
  - When you need a LIFO structure.
  - Generally when the most recent thing you've seen is most important or should be processed first.
  - E.g: basic parsers, dfs, bracket matching.
- Queues:
  - When you need a FIFO structure.
  - Generally when you want to process events in order of occurrence.
  - E.g: event processing, bfs.

- Heap:
  - When you find yourself asking how I can get the "largest/smallest" item.
  - E.g: Dijkstra's algorithm, other greedy algorithms.
- Set:
  - Seen array on unbounded keys. Also when you need to dynamically maintain a sort order.
  - E.g: Recognizing duplicates, find closest key to x.
- Map:
  - As above but with keyed data.
  - E.g: Count duplicates, find index of the closest key to x.

A warning

- While STL has a lot of nice functionality, it does have significant overhead. If your algorithm is of the correct time complexity but exceeds the time limit, you might achieve some constant factor speedup by removing unnecessary STL:
  - Replace vectors with arrays - allocate as much memory as you would ever need
  - Replace stacks and queues with arrays
  - Replace small sets with bitsets

Table of Contents

Recap: n countries, each with up to 20 delegates; m restaurants, each with up to 100 capacity

Recall this problem boiled down to:

- Process countries in any order.
- For each, seat delegates at restaurant with **most** seats, then second **most**, etc.
- Sounds like a max heap to me!

Example problem: Restaurants


```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

const int N = 2020, M = 2020;
int n, numDelegates[N], m;
priority_queue<int> restaurants;

int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> numDelegates[i];

    cin >> m;
    for (int i = 0; i < m; i++) {
        int s;
        cin >> s;
        restaurants.push(s);
    }
```

Example problem: Restaurants


```
    int starved = 0;
    for (int i = 0; i < n; i++) {
        vector<int> poppedRestaurants;
        int delegatesRemaining = numDelegates[i];

        while (delegatesRemaining && !restaurants.empty()) {
            // seat a delegate at the restaurant with the most seats.
            delegatesRemaining--;
            // remove this restaurant's capacity
            // to avoid seating multiple delegates here
            int seatsRemaining = restaurants.top();
            restaurants.pop();
            poppedRestaurants.push_back(seatsRemaining-1);
        }

        // only add back restaurants with positive remaining capacity
        // skip any that are now full
        for (int r : poppedRestaurants)
            if (r > 0)
                restaurants.push(r);

        // any unassigned delegates starve
        starved += delegatesRemaining;
    }
    cout << starved << '\n';
}
```

New complexity?

- Let A be the maximum number of delegates per country.
- $O(n \cdot A \cdot \log m) \approx 2000 \cdot 20 \cdot 11$, one hundred times faster!
- Efficiency comes from not re-sorting the entire list of $m \leq 2000$ restaurants every round.

Subarray Sum

- **Problem statement** You are given an array of n numbers, say $a_0, a_1, \ldots, a_{n-1}$. Find the number of pairs (i, j) with $0 \leq i < j \leq n$ such that the corresponding subarray satisfies

$$a_i + a_{i+1} + \ldots + a_{j-1} = S$$

  for some specified sum S.

- **Input** The size n of the array ($1 \leq n \leq 100,000$), and the n numbers, each of absolute value up to 20,000, followed by the sum S, of absolute value up to 2,000,000,000.

- **Output** The number of such pairs.

Subarray Sum

- **Algorithm 1** Evaluate the sum of each subarray, and if it equals S, increment the answer.
- **Complexity** There are $O(n^2)$ subarray, and each takes $O(n)$ time to add, so the time complexity is $O(n^3)$.
- **Algorithm 2** Compute the prefix sums

$$b_i = a_0 + a_1 + \ldots + a_{i-1}.$$

Then each subarray can be summed in constant time:

$$a_i + a_{i+1} + \ldots + a_{j-1} = b_j - b_i.$$

- **Complexity** This solution takes $O(n^2)$ time, which is an improvement but still too slow.

- We need to avoid counting the subarray individually.
- For each $1 \leq j \leq n$, we ask: how many $i < j$ have the property that $b_i = b_j - S$?
- If we know the frequency of each value among the b_i, we can add all the answers involving j at once.
- The values could be very large, so a simple frequency table isn't viable - use a map!

- **Algorithm 3** Compute the prefix sums as above. Then construct a map, and for each b_j, add the frequency of $b_j - S$ to our answer and finally increment the frequency of b_j.

- **Complexity** The prefix sums take $O(n)$ to calculate, since

$$b_{i+1} = b_i + a_i.$$

Since map operations are $O(\log n)$, and each b_j requires a constant number of map operations, the overall time complexity is $O(n \log n)$.

```
#include <iostream>
#include <map>
using namespace std;

const int N = 100100;
int a[N];
int b[N];

int main() {
  int n, S;
  cin >> n;
  // read input and compute prefix sums
  for (int i = 0; i < n; i++) {
    cin >> a[i];
    b[i+1] = b[i] + a[i];
  }
  cin >> S;

  // answer could be up to 100,000 choose 2, approx 5e9
  long long ret = 0;
  map<int,int> freq;
  // freq[x] = k means that k of the prefix sums found so far equal x
```

```
  // for each endpoint
  for (int j = 0; j <= n; j++) {
    /* each start point i
       satisfying b[i] = b[j] - S
       contributes 1 to the answer */
    /* if b[j] - S isn't already a key in the map
       it will be created with value 0, which is fine */
    ret += freq[b[j]-S];

    /* now add b[j] itself to the map
       as future endpoints should consider index j as a start point */
    freq[b[j]]++;
  }

  cout << ret << '\n';
}
```

Largest Interval

- **Problem statement** You have $M \leq 1,000,000,000$ chairs, initially all empty. There are $U \leq 100,000$ updates, in each a person comes in and takes an unoccupied chair c_i. After each update, what is the longest range of unoccupied chairs?
- **Input** First line, M then U. Next U lines, each contains one integer, $1 \leq c_i \leq M$. Guaranteed no integer appears more than once.
- **Output** For each update, an integer, the longest range of unoccupied chairs after the update.

Largest Interval

- **Sample Input:**

```
12 3
5
7
10
```

- **Sample Output:**

```
7
5
4
```

- **Observation 1:** We only care about maximal ranges. Assuming chair 0 and chair $M + 1$ are occupied, we only care about ranges starting and ending with occupied chairs.
- So we will maintain for each chair, what is the length of the range to its right.
- How does an update change the intervals?
- It breaks one apart and adds 2.

Largest Interval

- What data do we need to store to handle updating intervals (i.e: to determine what the 2 new intervals are when we insert a chair)?
- For each update, we need to find the closest chair in both directions.
- We need to maintain a *sorted* list of chairs *associating* with each chair the length of the range starting at that chair.
- Map!
- Figuring out the new range lengths is basic maths, just be careful with off-by-1s!

Largest Interval

- Now we know how to track length of each range. Remains to track the *largest* of the ranges.
- Heap!
- But wait, heaps can not do arbitrary deletions ...(which we need when we delete an interval)
- Set!

```
#include <iostream>
#include <map>
#include <set>
using namespace std;

int M, U;
map<int, int> chairToRange;
multiset<int> allRanges;

// insert a new chair at 'start' with range 'length'
void addRange(int start, int length) {
    chairToRange[start] = length;
    allRanges.insert(length);
}

// update an existing chair to range 'length'
void updateRange(int start, int length) {
    int oldLength = chairToRange[start];
    chairtoRange[start] = length;
    // allRanges.erase(val) erases all entries of value val
    // instead, get an iterator to one instance of the old length
    // this deletes just one copy
    allRanges.erase(allRanges.find(oldLength));
    allRanges.insert(length);
}
```

```
int main() {
    cin >> M;
    // insert dummy occupied chairs at each end, to avoid special cases
    addRange(0, M);
    addRange(M+1, 0);

    cin >> U;
    for (int i = 0; i < U; i++) {
        int q;
        cin >> q;
        // find first map entry whose key compares >= q
        // *it is a pair of (first chair right of q, range length)
        auto it = chairToRange.lower_bound(q);
        // length of empty range right of new chair
        int qLength = it->first - q - 1;
        // now access chair left of q
        --it;
        // existing range from this chair must be shortened
        int updatedLength = q - it->first - 1;

        addRange(q, qLength);
        updateRange(it->first, updatedLength);

        // s.rbegin() returns an iterator to the last (i.e. biggest) entry
        cout << *allRanges.rbegin() << '\n';
    }
    return 0;
}
```

- Many of our data structures work best if data is sorted.
- E.g: we can then chuck them into a set and use `lower_bound`
- Or we can chuck them into a vector and binary search.
- Sometimes we have to work a bit to get this!

Area Under Histogram

- **Problem statement** Given a histogram with n unit-width columns, the i-th with height h_i. What is the largest area of a rectangle that fits under the histogram.

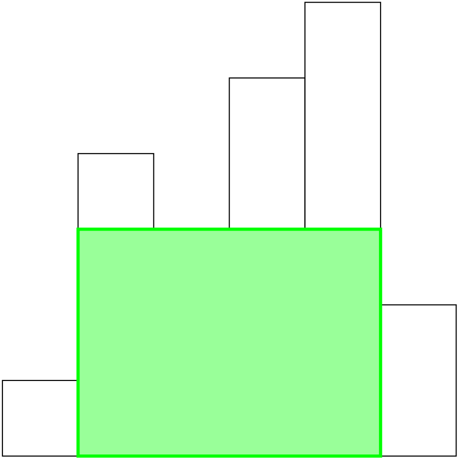
- **Input** The integer $1 \leq n \leq 100,000$ and n numbers, $0 \leq h_i \leq 1,000,000,000$.

- **Output** The largest area of a rectangle that you can fit under the histogram.

- **Sample Input:**

  6

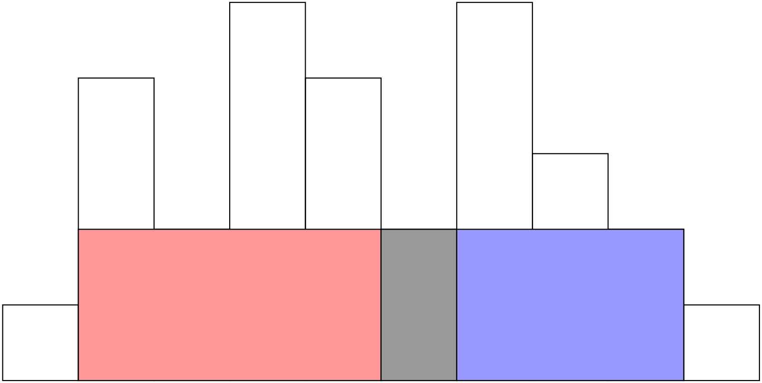
  1 4 3 5 6 2
- **Sample Output:** 12



Area Under Histogram

- **Observation 1:** We only care about "maximal" rectangles.
- More formally, they hit some column's roof and can not be extended to the left or right.
- Many angles to approach this problem. Let us focus on one specific column's roof. We now want to find the largest histogram that hits that column's roof.
- **Claim:** We just need to know the first column to its left (and right) that has lower height than it.
- But we need this for all choices of our "specific column". So we will try to do this in a linear sweep and maintain some sort of data structure that can answer this.

Area Under Histogram



$$\text{area}[i] = \text{h}[i] \times (\text{left}[i] + 1 + \text{right}[i])$$

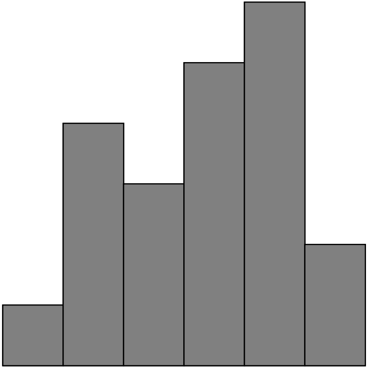
Area Under Histogram

- Queries: what is the first column with height $< h$.
- Updates: add a new column (pos, h_{pos}) where pos is greater than all previous positions.
- Multimap? But what can we search on...?
  - If our key is height then we can find a column lower than us. But it is not guaranteed to be the closest one.
  - If our key is position then we can't do anything.
- Heap? Again, same problem (our heap can't do anything a set can't do).

- **Key Observation:** Out of all added columns, we only care about columns that have no lower columns to their right!

- So if we only keep these columns in our map, then the first column in our map lower than us is also the closest column lower than us.

Area Under Histogram



i	0	1	2	3	4	5
left[i]	0	0	1	0	0	4

Area Under Histogram

- **Complexity?** Dominated by map operations.
- $O(n)$ calls to `lower_bound` and `insert`.
- Each column can only be removed from the map once, so $O(n)$ calls to `erase` also.
- Total time to sweep left to right is $O(n \log n)$ (amortised $O(\log n)$ per column).
- Repeat this right to left, and add a bit of maths to solve original problem regarding largest rectangle under histogram.

```
#include <iostream>
#include <map>
using namespace std;

const int N = 100100;
int h[N];

// left[i] is number of columns immediately left of i with height >= h[i]
// i.e. how far left can you stretch bar i without going outside the histogram
int left[N], right[N];

void sweepltor(); // computes all left[i] values in O(n log n)
void sweeprtol(); // computes all right[i] values in O(n log n)

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> h[i];

    sweepltor();
    sweeprtol(); // left as an exercise

    long long ans = 0;
    for (int i = 0; i < n; i++) {
        // area of rectangle formed by stretching bar i left and right
        long long cur = 1LL * (left[i] + 1 + right[i]) * h[i];
        ans = max(ans, cur);
    }
    cout << ans << '\n';
}
```

Area Under Histogram

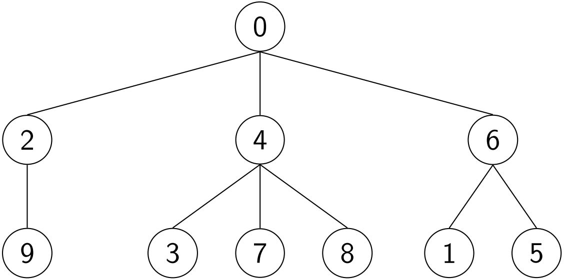

```
void sweepltor {
    // height -> column index
    map<int, int> importantColumns;

    // dummy leftmost bar at index -1 with height -2
    // must have lower height than all actual bars
    importantColumns[-2] = -1;

    for (int i = 0; i < n; i++) {
        // find closest column to i's left with lower height
        // lower_bound finds first >= h[i]
        // so prev(lower_bound) finds last < h[i]
        auto it = prev(importantColumns.lower_bound(h[i]));

        // left[i] counts columns strictly between this column and i
        left[i] = i - it->second - 1;

        /* some columns might no longer be important
           as a result of column i being both later and shorter */
        while (importantColumns.rbegin()->first >= h[i])
            // m.erase(it) requires forward iterator
            // instead erase by key
            importantColumns.erase(importantColumns.rbegin()->first);

        // column i is important (at least for now)
        importantColumns[h[i]] = i;
    }
}
```

- Each sweep could be sped up to $O(n)$ by using a stack instead of a map.
  - Inserting (pushing) and deleting (popping) go from $O(\log n)$ to $O(1)$.
  - Popping the stack before calculating `left[i]` makes the binary search unnecessary.
- **Challenge:** There is a beautiful algorithm that does it in one stack sweep in $O(n)$. Essentially the same idea except process a rectangle not at the column where it attains its maximum but at the right end.
- Another famous problem using a similar idea is Longest Increasing Subsequence.

Rooted Trees

- A tree is a connected, undirected graph with a unique simple path between any two vertices.
- A *rooted* tree is one with a designated root.
  - All other vertices have a parent $par[v]$, which is the next node in the unique path from v to the root.
- An easy way to represent a rooted tree is to just store this parent array.

Rooted Trees



v	0	1	2	3	4	5	6	7	8	9
$par[v]$	0	6	0	4	0	6	0	4	4	2

Also called a *system of disjoint sets*; used to represent disjoint sets of items.

Given some set of elements, support the following operations:

- union(x, y): union the disjoint sets that contain x and y
- find(x): return a canonical representative for the set that x is in
  - More specifically, we must have $\text{find}(x) = \text{find}(y)$ whenever x and y are in the same set.
  - It is okay for this answer to change as new elements are joined to a set. It just has to remain consistent across all elements in each disjoint set at a given moment in time.

- **Strategy:** Represent each disjoint set as a rooted tree. The representative of each rooted tree is the chosen root. For this, we just need to store the parent of each element.
- For find(x), walk up parent edges from x until a root (a vertex who is their own parent) is found
- For union(x, y), add an edge between the trees containing x and y
  - When the two trees are joined, what's the new root?
  - Don't add the edge between x and y directly
  - Instead add the edge between find(x) and find(y), and designate find(x) as the new parent of find(y)

Union-Find: Basic Implementation

- Both operations require us to walk $O(h)$ edges, where h is the height of the tree
- In the worst case, both operations take $O(n)$

```
int parent[N];

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

int root(int x) {
  // only roots are their own parents
  return parent[x] == x ? x : root(parent[x]);
}

void join(int x, int y) {
  // join roots
  x = root(x); y = root(y);
  // test whether already connected
  if (x == y)
    return;
  parent[y] = x;
}
```

- In the basic implementation, we made find(x) the new parent of find(y), but the inverse would also be valid. Which one is better?
- We should hang the *smaller* subtree from the root of the *larger* subtree
- The maximum height of the tree is now $O(\log n)$
  - When we traverse the edges from any particular element to its parent, we know that the subtree rooted at our current element must at least double in size, and we can double in size at most $O(\log n)$ times
- Therefore **find** and **union** now take only $O(\log n)$ time

Union-Find: Size Heuristic


```
int parent[N];
int subtree_size[N];

void init(int n) {
  for (int i = 0; i < n; i++) {
    parent[i] = i;
    subtree_size[i] = 1;
  }
}

int root(int x) {
  // only roots are their own parents
  return parent[x] == x ? x : root(parent[x]);
}

void join(int x, int y) {
  // join roots
  x = root(x); y = root(y);
  // test whether already connected
  if (x == y)
    return;
  // size heuristic
  // hang smaller subtree under root of larger subtree
  if (subtree_size[x] < subtree_size[y]) {
    parent[x] = y;
    subtree_size[y] += subtree_size[x];
  } else {
    parent[y] = x;
    subtree_size[x] += subtree_size[y];
  }
}
```

Union-Find: Path Compression

- When performing a find operation on some element x, instead of just returning the representative, we change the parent edge of x to whatever the representative was, flattening that part of the tree
- This optimisation alone gives an amortised $O(\log n)$ per operation complexity. Proof is nontrivial, omitted.

```
int root(int x) {
  // only roots are their own parents
  // otherwise apply path compression
  return parent[x] == x ? x : parent[x] = root(parent[x]);
}
```

- Combined with the size heuristic, we get a time complexity of amortised $O(\alpha(n))$ per operation, but the proof is very complicated.
- $\alpha(n)$ is the inverse Ackermann function, a very slow growing function which is less than 5 for $n < 2^{2^{2^{2^{16}}}}$.
- As mentioned, the above two optimisations together bring the time complexity down to amortised $O(\alpha(n))$.

- **Warning:** due to a low-level detail, the path compression optimisation actually significantly slows down the find function, because we lose the tail recursion optimisation, now having to return to each element to update it.

- This may overshadow the improvement from $O(\log n)$ to $O(\alpha(n))$, depending on bounds of the problem.

Example problem: Dynamic Connectivity

The main application of union find.

Given a graph with $n \leq 100,000$ vertices and no edges, support $m \leq 100,000$ operations of two forms.

- update
  - denoted `U a b`
  - add an undirected edge between a and b
- query
  - denote `Q a b`
  - output 1 if a and b are connected, 0 otherwise.

Example problem: Dynamic Connectivity


```
#include <iostream>
using namespace std;

// insert your union find implementation here
int root (int x);
void join (int x, int y);

int main() {
    int n, m;
    cin >> n >> m;
    for (int q = 0; q < m; q++) {
        char queryType;
        int a, b;
        cin >> queryType >> a >> b;
        if (queryType == 'U')
            join(a,b);
        else
            cout << (root(a) == root(b)) << '\n';
    }
    return 0;
}
```

- **When is it useful?** When you need to maintain which items are in the same set.

- **Main limitation:** You **can not** delete connections, only add them. However, in a lot of natural contexts, this is not a restriction since items in the same set can be treated as the same item.

Table of Contents

- Last main topic is data structures that support operations on a range
- Why do we care about this?
- Pragmatic answer: impossible to just support arbitrary queries and updates, but there is a lot of interesting stuff we can do with ranges.
- But also, naturally applies to many problems, e.g: ranges of numbers, a range in an array, linear sweeps often result in caring about ranges ...

- **Problem:** Given n integers $a_0, a_1, \ldots, a_{n-1}$, answer q queries of the form

$$\sum_{i=l}^{r-1} a_i$$

for given pairs l, r.

- $n, q \leq 100,000$
- **Attempt 1** Store the a_i in an array and answer queries naïvely.
- **Complexity** Each query could take $O(n)$, so the complexity is $O(nq)$, which is too slow.
- Instead, we need to do some kind of precomputation.

Prefix Sums

- **Algorithm** Construct an array of prefix sums, using (rudimentary) dynamic programming.
- Base case: $b_0 = a_0$.
- Recurrence: $b_i = b_{i-1} + a_i$.
- This takes $O(n)$ time.
- Now, we can answer every query in $O(1)$ time, so the total complexity is $O(n + q)$.

- This works on any “reversible” operation. That is, any operation $A \star B$ where if we know $A \star B$ and A, we can find B.

- This includes addition and multiplication, but *not* max or gcd.

- There is also a 2D analogue: see the tutorial problem Quality of Living.

Range Max Queries

- **Problem** Given n integers $a_0, a_1, \ldots, a_{n-1}$, answer q queries of the form

$$\max a[l, r)$$

  for given pairs l, r.

- $n, q \leq 100,000$.

- Again, the naïve approach answers each query in $O(n)$, so we need to do some kind of precomputation instead.

- Prefix max is unhelpful: knowing $\max a[0, l)$ and $\max a[0, r)$ says almost nothing about $\max a[l, r)$.

- Max is not "reversible" but it does have a different property: *idempotence*. This just means $\max(x, x) = x$, i.e: I can apply max as many times as I want to the same element, it does not do anything.
- It's therefore sufficient to cover a range $[l, r)$ with two intervals $[l, s)$ and $[t, r)$ that *may overlap*.
- If $l \leq s \leq t \leq r$ then

$$\max a[l, r) = \max(\max a[l, t), \max a[s, r))$$

- So we want to precompute the max of a bunch of intervals, such that any subarray $a[l, r)$ can be written as the union of two of these intervals.

- **Key Idea:** Precompute the max of all intervals whose lengths are powers of 2.

- This can be done quickly since an interval of length 2^k is the union of two intervals of length 2^{k-1}.

Sparse Tables: Precomputation Implementation


```
const int N = 100000;
const int LOGN = 18;

int a[N];
// sparseTable[l][i] = max a[i..i+2^l)
int sparseTable[LOGN][N];

void precomp(int n) {
    // level 0 is the array itself
    for (int i = 0; i < n; i++)
        sparseTable[0][i] = a[i];

    for (int l = 1; l < LOGN; l++) { // inner loop does nothing if 2^l > n
        int w = 1 << (l-1); // 2^(l-1)

        // a[i,i+2w) is made up of a[i,i+w) and a[i+w,i+2w)
        for (int i = 0; i + 2*w <= n; i++)
            sparseTable[l][i] = max(sparseTable[l-1][i],sparseTable[l-1][i+w]);
    }
}
```

Sparse Tables

- Suppose we now want $\max a[l, r)$.
- Let $p = 2^k$ be the largest power of 2 that is $\leq r - l$.
- **Key Observation:** since $p \leq r - l < 2p$, we have

$$l \leq r - p < l + p \leq r,$$

i.e. $[l, l + p)$ and $[r - p, r)$ cover $[l, r)$ (with overlap).

- **Hence:**

$$\max a[l, r) = \max(\max a[l, l + p), \max a[r - p, r))$$

- But both intervals on the RHS have length a power of 2, so we have precomputed them!

Sparse Tables: Query Implementation


```
#include <algorithm>
#include <iostream>
using namespace std;

const int N = 100000, LOGN = 18;

int a[N], sparseTable[LOGN][N];
int log2s[N];

void precomp(int n);

int main() {
    // Input the initial array
    int n; cin >> n;
    for (int i = 0; i < n; i++)
        cin >> a[i];
    precomp(n);

    // log2s[i] = floor(log_2(i))
    for (int i = 2; i <= n; i++)
        log2s[i] = log2s[i/2] + 1;

    int q; cin >> q;
    for (int j = 0; j < q; j++) {
        int l, r; cin >> l >> r;
        // Problem: Find max of a[l,r)
        int lvl = log2s[r-l];
        cout << max(sparseTable[lvl][l], sparseTable[lvl][r-(1<<lvl)]) << '\n';
    }
}
```

- **Complexity?** $O(n \log n)$ precomp, $O(1)$ per query.
- **Warning:** You need your operation to be idempotent. This will double count for sum, multiply, count, etc ...
- Works for $\max, \min, \gcd, \text{lcm}$.
- Practically, don't see it too often. But a nice idea, and the data structure for Lowest Common Ancestor is similar.

Range Sum with Updates

Prefix sums and sparse tables do not support updates.

- **Problem** Given n integers $a_0, a_1, \ldots, a_{n-1}$, answer q queries of the form

$$\sum_{i=l}^{r-1} a_i$$

  for given pairs l, r.

- **But** there are now also u updates of the form "set $a_i = k$".
- $n \leq 100,000,\ q + u \leq 100,000$.

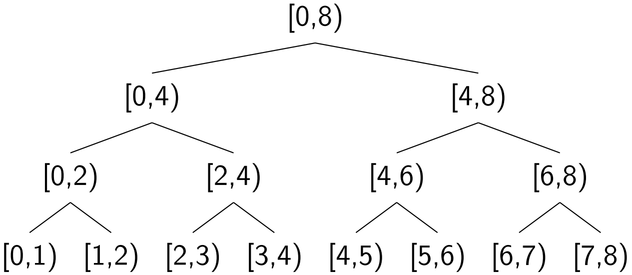
Problem: Updates

- Recomputing the prefix sums will take $O(n)$ time per update, so our previous solution is now $O(n^2)$ for this problem, which is too slow.
- We don't need to answer queries in constant time; it just needs to be much faster than linear.
- Let's try to find a compromise that slows down our queries but speeds up updates in order to improve the overall complexity.

- The problem with prefix sums (and sparse tables) is there are too many ranges containing any given value, so updating all of them is $O(n)$ per update.
- The problem with just storing the array is that any subarray might need to be comprised from many ranges, so querying is $O(n)$.
- We need to decompose $[0, n)$ into ranges such that:
  - each item belongs to much fewer than n ranges, and also
  - any subarray can be decomposed into much fewer than n ranges.

Let's make a tree

- We will make a tree. Each node in the tree is responsible for a range.

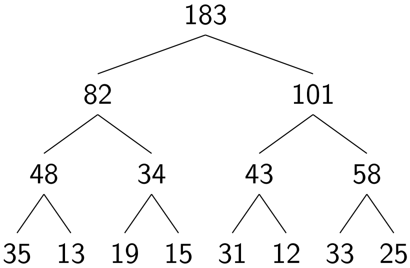


- The array itself goes into the leaves.
- The internal nodes store information on the range, depending on the problem at hand.
- For our earlier problem, we would want each node to store the sum over its range of responsibility.

- Consider the array

$$[35, 13, 19, 15, 31, 12, 33, 23]$$

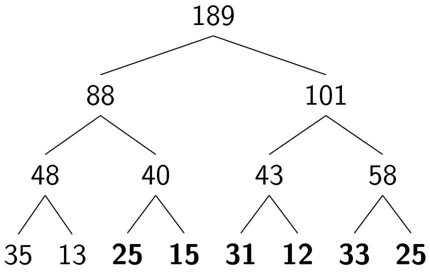
- We would get the tree



- Note that the leaves store the array, and every other node is just the sum of its two children.

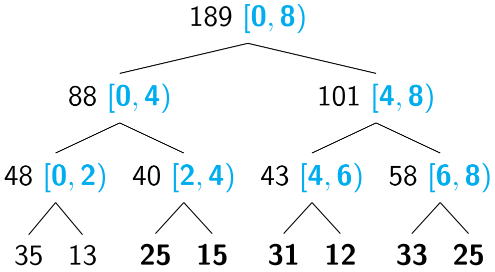
Queries

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



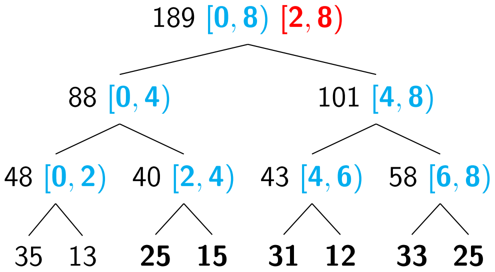
- Recall each node in the tree has a "range of responsibility".

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



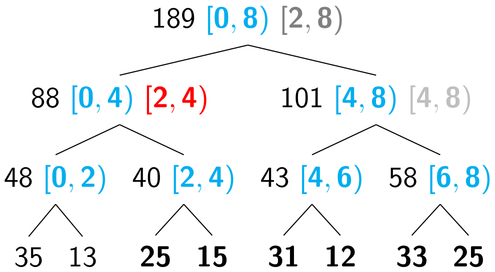
- Our goal is the same as in the sparse table: find a set of ranges whose disjoint union is $[2, 8)$. Then taking the sum of those nodes gives us the answer.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



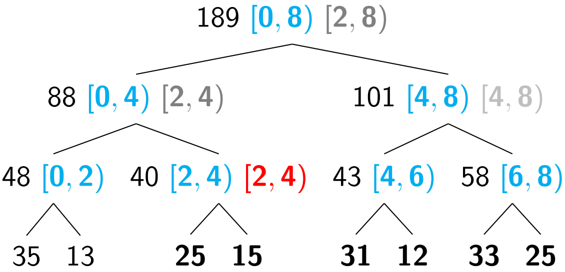
- We start at the top of the tree, and 'push' the query range down into the applicable nodes.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



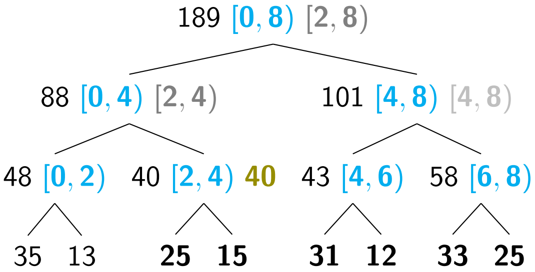
- This is a recursive call, so we do one branch at a time. Let's start with the left branch.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



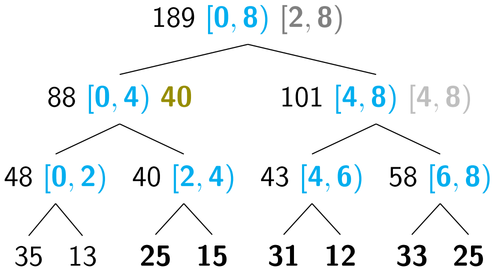
- There is no need to continue further into the left subtree, because it doesn't intersect the query range.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



- There is also no need to continue further down, because this range is equal to our query range.

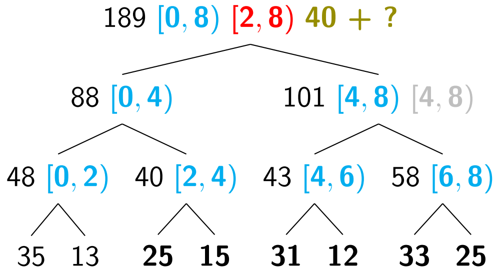
- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



- We return the result we have obtained up to the chain, and let the query continue.

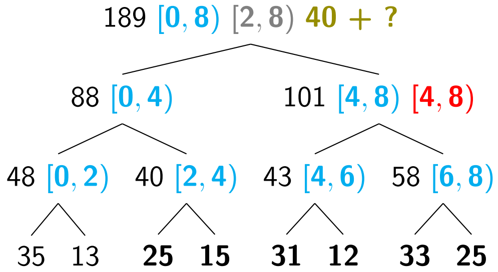
Queries

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



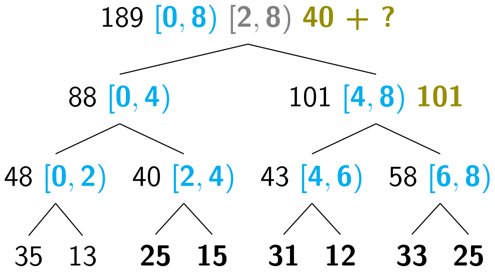
- We return the result we have obtained up to the chain, and let the query continue.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



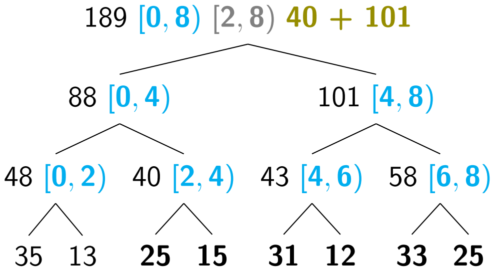
- Now, it is time to recurse into the other branch of this query.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



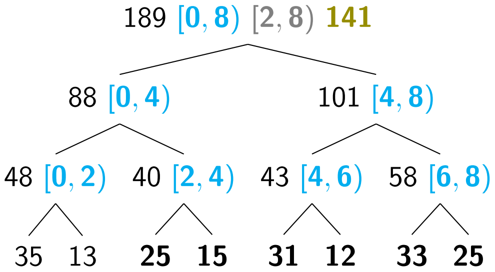
- Here, the query range is equal to the node's range of responsibility, so we're done.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



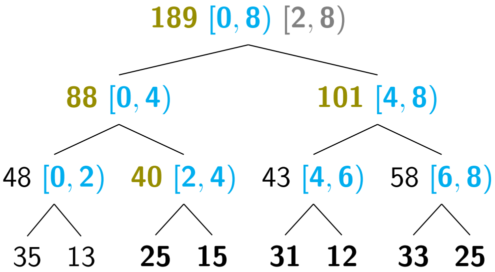
- Here, the query range is equal to the node's range of responsibility, so we're done.

- Let's query the sum of $[2, 8)$ (inclusive-exclusive).



- Now that we've obtained both results, we can add them together and return the answer.

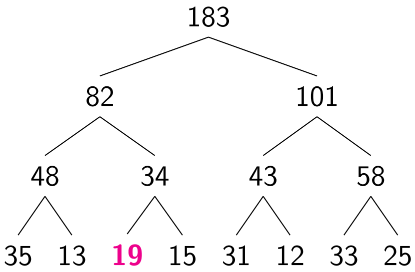
- We didn't visit many nodes during our query.



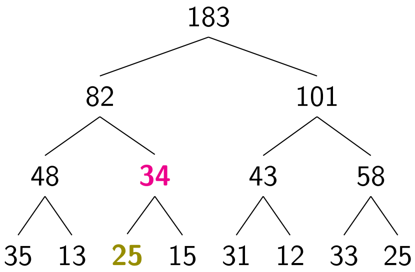
- In fact, because only the left and right edges of the query can ever get as far as the leaves, and ranges in the middle stop much higher, we only visit $O(\log n)$ nodes during a query.

- One way to see this is consider cases based on if the query range shares an endpoint with the current node's range of responsibility.
- Another way is to consider starting with the full range from the bottom and going up.
- Probably easiest if you play around a bit and convince yourself of this fact.

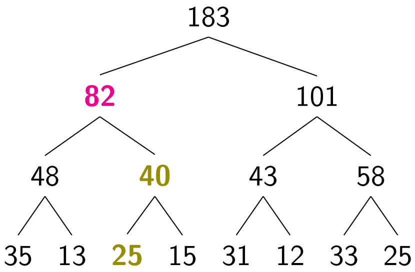
- Let's update the element at index 2 to 25.



- Let's update the element at index 2 to 25.

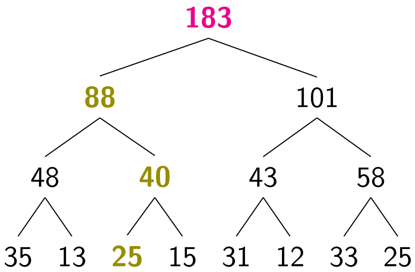


- Let's update the element at index 2 to 25.

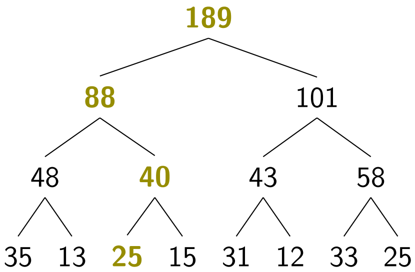


Updates

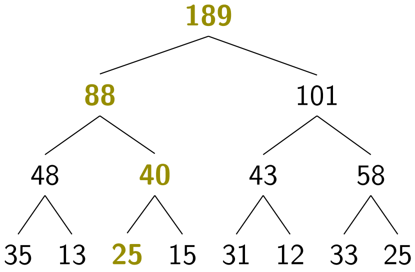
- Let's update the element at index 2 to 25.



- Let's update the element at index 2 to 25.



- Let's update the element at index 2 to 25.



- We always construct the tree so that it's balanced, i.e. its height is approximately $\log n$.
- Thus, updates take $O(\log n)$ time.

- Thus we have $O(\log n)$ time for both updates and queries.
- This data structure is commonly known as a range tree, segment tree, interval tree, tournament tree, etc.
- The number of nodes we add halves on each level, so the total number of nodes is still $O(n)$.

- For ease of understanding, the illustrations used a *full* binary tree, which always has a number of nodes one less than a power-of-two.
- This data structure works fine as a *complete* binary tree as well (all layers except the last are filled).
  - This case is harder to imagine conceptually but the implementation works fine.
  - For each internal node just split the range of responsibility at its midpoint.
- All this means is that padding out the data to the nearest power of two is not necessary.

- Since these binary trees are complete, they can be implemented using the same array-based tree representation as with an array heap
  - Place the root at index 0. Then for each node i, its children (if they exist) are $2i + 1$ and $2i + 2$.
  - Alternatively, place the root at index 1, then for each node i the children are $2i$ and $2i + 1$.
- This works with any binary associative operator, e.g.
  - sum
  - min or max
  - gcd or lcm
  - merge (from merge sort)
    - For a non-constant-time operation like this one, multiply the complexity of all range tree operations by the complexity of the merging operation.

```
const int N = 100100;
// the number of additional nodes created can be as high as the next power of
      two up from N (2^17 = 131,072)
int tree[1<<18];

int n; // actual length of underlying array

// query the sum over [qL, qR) (0-based)
// i is the index in the tree, rooted at 1 so children are 2i and 2i+1
// instead of explicitly storing each node's range of responsibility [cL,cR), we
       calculate it on the way down
// the root node is responsible for [0, n)
int query(int qL, int qR, int i = 1, int cL = 0, int cR = n) {
  // the query range exactly matches this node's range of responsibility
  if (cL == qL && cR == qR)
    return tree[i];
  // we might need to query one or both of the children
  int mid = (cL + cR) / 2;
  int ans = 0;
  // query the part within the left child [cL, mid), if any
  if (qL < mid) ans += query(qL, min(qR, mid), i * 2, cL, mid);
  // query the part within the right child [mid, cR), if any
  if (qR > mid) ans += query(max(qL, mid), qR, i * 2 + 1, mid, cR);
  return ans;
}
```

Range Tree: Update Implementation


```
// p is the index in the array (0-based)
// v is the value that the p-th element will be updated to
// i is the index in the tree, rooted at 1 so children are 2i and 2i+1
// instead of explicitly storing each node's range of responsibility [cL,cR), we
      calculate it on the way down
// the root node is responsible for [0, n)
void update(int p, int v, int i = 1, int cL = 0, int cR = n) {
  if (cR - cL == 1) {
    // this node is a leaf, so apply the update
    tree[i] = v;
    return;
  }
  // figure out which child is responsible for the index (p) being updated
  int mid = (cL + cR) / 2;
  if (p < mid)
    update(p, v, i * 2, cL, mid);
  else
    update(p, v, i * 2 + 1, mid, cR);
  // once we have updated the correct child, recalculate our stored value.
  tree[i] = tree[i*2] + tree[i*2+1];
}
```

Range Tree: Debug Implementation


```
// print the entire tree to stderr
// instead of explicitly storing each node's range of responsibility [cL,cR), we
      calculate it on the way down
// the root node is responsible for [0, n)
void debug(int i = 1, int cL = 0, int cR = n) {
  // print current node's range of responsibility and value
  cerr << "tree[" << cL << "," << cR << ") = " << tree[i];

  if (cR - cL > 1) { // not a leaf
    // recurse within each child
    int mid = (cL + cR) / 2;
    debug(i * 2, cL, mid);
    debug(i * 2 + 1, mid, cR);
  }
}
```

- It is possible to construct a range tree in $O(n)$ time, but anything you use it for will take $O(n \log n)$ time anyway.
- Instead of writing extra code to construct the tree, just call `update()` repeatedly for $O(n \log n)$ construction.

- We can extend range trees to allow range updates in $O(\log n)$ using *lazy propagation*
- The basic idea is similar to range queries: push the update down recursively into the nodes whose range of responsibility intersects the update range.
- However, to keep our $O(\log n)$ time complexity, we can't actually update every value in the range.
- Just like we returned early from queries when the query range matched a node's entire range, we cache the update at that node and return without actually applying it.
- When a query or a subsequent update is performed which visits this node you might need to push the cached update one level further down.
- Will talk more about this in later lectures (Data Structures II).

- **Problem statement** Given an array of integers, find the maximum length of a (strictly) increasing (not necessarily contiguous) subsequence.
- **Input** An integer n, the size of the array, followed by n integers, a_i. $1 \leq n, a_i \leq 100,000$.
- **Output** A single integer, the length of the longest increasing subsequence.

- **Example Input**

  ```
  5
  4 0 3 2 8
  ```

- **Example Output** 3

- **Explanation**: Both $0, 3, 8$ and $0, 2, 8$ are longest increasing subsequences.

- We will compute this iteratively using dynamic programming.
- For each index, let best[i] be the length of the longest increasing subsequence ending at index i.
- How do we compute best[i]? Either it's 1 or it extends an existing subsequence; in particular, the longest subsequence ending at some earlier index j containing a smaller array entry.
- Recurrence:

$$\text{best}[i] = 1 + \max\{\text{best}[j] \mid j < i, a[j] < a[i]\}.$$

- Recurrence:

$$\text{best}[i] = 1 + \max\{\text{best}[j] \mid j < i, a[j] < a[i]\}.$$

- Direct implementation runs in $O(n)$ per index, i.e. $O(n^2)$ total; too slow.
- The restriction $j < i$ is handled by the sweep order; only consider best[j] values seen so far.
- But we can't just keep the largest of the best[j] values seen so far, because we have to filter only those where $a[j] < a[i]$.

- So we want to query, over all j where $a[j] < a[i]$, what is the **max** value of best[j].
- This looks like a range query. But over a range of *what*?
- **Solution:** Range tree indexed by the values $a[j]$!

- Let $\text{bestWithEnd}[h]$ be the length of the longest subsequence ending at *value* h.
  - Define $\text{bestWithEnd}[0] = 0$ to avoid special cases.
- As we sweep, we maintain a range tree over this array. When we get to a new index i,
  - we record that the longest subsequence ending at *index* i has length

$$\text{best}[i] = 1 + \max \text{bestWithEnd}[0, a[i]),$$

  - and we update $\text{bestWithEnd}[a[i]]$ with this $\text{best}[i]$ value.
- The final answer is the length of the longest subsequence to finish at *any index*, or equivalently at *any value*, so it too can be found by a range query.

```
#include <algorithm>
#include <iostream>
using namespace std;

const int N = 100100;
int tree[1<<18];

// range max tree over array values (not indices)
// note: root covers [0,N) not [0,n)
int query(int qL, int qR, int i = 1, int cL = 0, int cR = N);
void update(int p, int v, int i = 1, int cL = 0, int cR = N);

int main() {
  int n;
  cin >> n;

  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    int best = 1 + query(0, x);
    update(x, best);
  }

  cout << query(0, N) << '\n';
}
```

- **Complexity?** $O(n)$ range tree queries and updates, each $O(\log n)$. Total: $O(n \log n) \approx 100,000 \cdot 17$.
- **Moral:** When trying to solve a problem, be on the lookout for suboperations that might be sped up by data structures. Often take the form of needing to support simple queries.
- Also it is useful to consider range trees over values, not just indices.
- The bound $h_i \leq 100,000$ was not necessary; only the relative order of the h_i values mattered. So we could have sorted them and replaced each with its rank in the resulting array - "coordinate compression".

- Alternatively, instead of doing it from left to right, one can solve it in increasing order of values in the array. Then your range tree is over indices not values, and your queries become "what is the largest value in best$[0, i)$".
- There is also a really elegant solution with a left to right sweep and a sorted stack. Let minEnd[i] store the minimum end value for a subsequence of length i. This is a sorted array (prove it) and we can update it in $O(\log n)$ time with binary search.

Example problem: Card Trick

- **Problem statement** Magnus the Magnificent is a magician. In his newest trick, he places n cards face down on a table, and turns to face away from the table. He then invites q members of the audience to do either of the following moves:

  1. announce two numbers i and j, and flip all cards between i and j inclusive, or
  2. ask him whether a particular card k is face up or face down.

  Unfortunately, Magnus the Magnificent is unable to do this trick himself, so write a program to help him!

- **Input** The numbers n and q, each up to 100,000, followed by q lines either of the form `F i j` ($1 \leq i \leq j \leq n$), a flip, or `Q k` ($1 \leq k \leq n$), a query.
- **Output** For each query, print "Face up" or "Face down".

Example problem: Card Trick

- Observe that we can just keep track of how many times each card was flipped; the parity of this number determines whether it is face up or face down.
- The operations appear to be range updates and point queries.
- We know how to do *point* updates and *range* queries; can we make these operations fit our existing framework?

Example problem: Card Trick

- **Idea** Don't store the number of times card k has been flipped directly. Instead store enough information so that you can *quickly calculate* the number of flips containing card k.
- Handle a flip $[i, j]$ by adding 1 at the left endpoint and subtracting 1 immediately after the right endpoint.
- Now, the sum over the first k cards is the number of times that card k has been flipped!

Example problem: Card Trick

- **Algorithm** Construct a range tree.
  - For the move `F i j`, increment a_{i-1} and decrement a_j.
  - For the move `Q k`, calculate $a_0 + a_1 + \ldots + a_{k-1}$ modulo 2.
  - Note the conversion to 0-based indexing.
- **Complexity** Each of these operations takes $O(\log n)$ time, so the time complexity is $O(q \log n)$.

Example problem: Card Trick


```
#include <iostream>
using namespace std;

const int N = 100100;
int tree[1<<18];

int n;

// range sum tree
int query(int qL, int qR, int i = 1, int cL = 0, int cR = n);
void update(int p, int v, int i = 1, int cL = 0, int cR = n);

int main() {
  int q;
  for (int i = 0; i < q; i++) {
    char type;
    cin >> type;
    if (type == 'F') {
      int i, j;
      cin >> i >> j;
      update(i-1, 1);
      update(j, -1);
    }
    else if (type == 'Q') {
      int k;
      cin >> k;
      cout << ((query(0, k) % 2) ? "Face up\n" : "Face down\n");
    }
  }
}
```