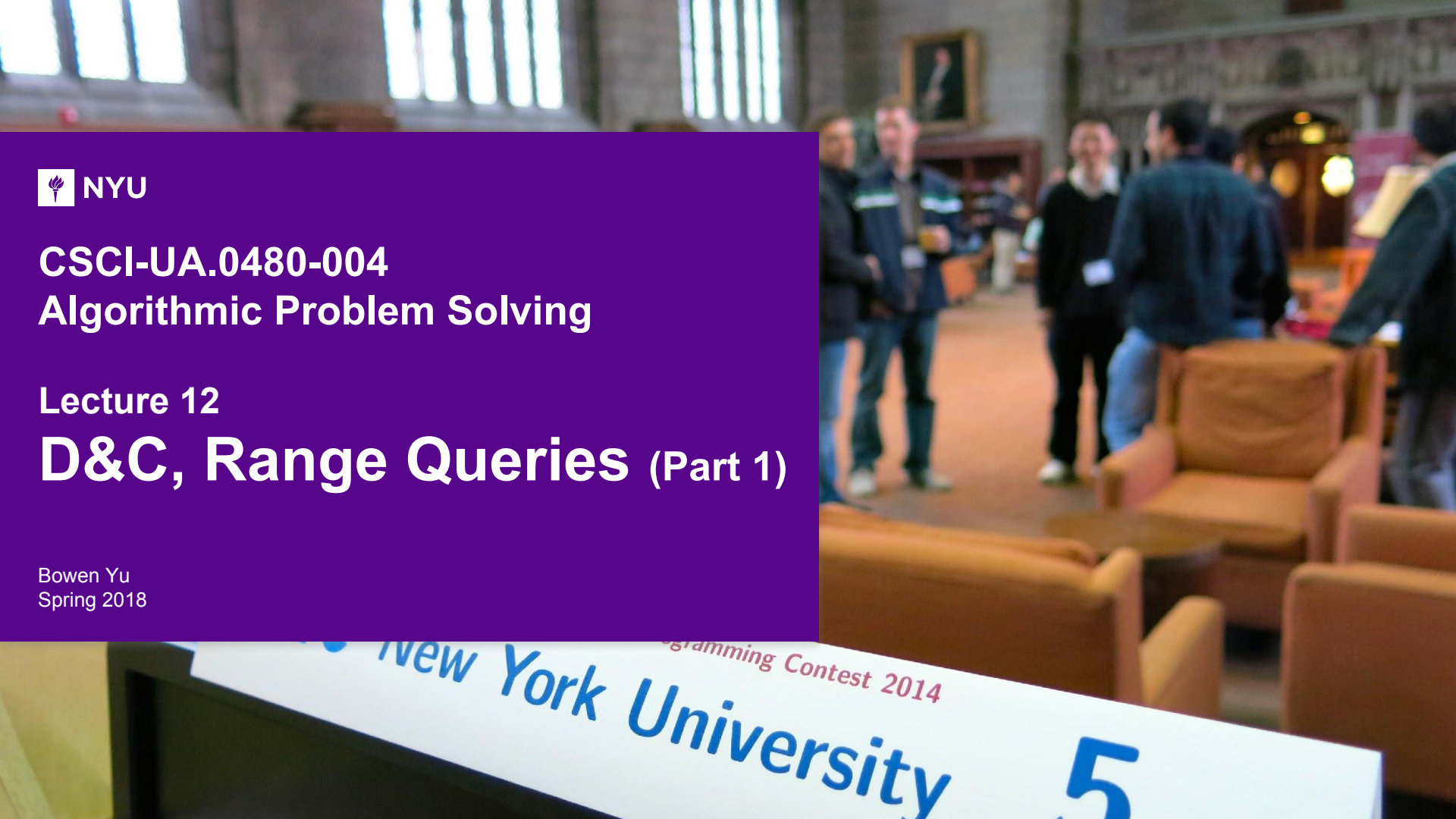


NYU

CSCI-UA.0480-004
Algorithmic Problem Solving

Lecture 12
D&C, Range Queries (Part 1)

Bowen Yu
Spring 2018

- HW7
  - Complete Search & Greedy I
  - Due on Sunday

- HW8 Released
  - Complete Search & Greedy II
  - Due on next Wednesday

Friends

Each person has a node in the union find. When merging two groups, let the new group's root store the youngest and eldest persons.

```
1  void unionNodes(Node x, Node y) {
2    Node xRoot = findRoot(x), yRoot = findRoot(y);
3    if (xRoot.rank > yRoot.rank)
4      swap(xRoot, yRoot);
5    if (xRoot.rank == yRoot.rank)
6      yRoot.rank += 1;
7    xRoot.parent = yRoot;
8
9    if (younger(xRoot.youngest, yRoot.youngest)) yRoot.youngets = xRoot.youngest;
10   if (elder(xRoot.eldest, yRoot.eldest)) yRoot.eldest = xRoot.eldest;
11 }
```

Time: O(nL), where L=10 is the max name length, i.e. the time to update youngest and eldest.

Red, Black Cards (Solution 1)

Create two extra cards Red and Black.

- r x: union(x, Red), union(x', Black)
- b x: union(x, Black), union(x', Red)
- s x y: union(x, y), union(x', y')
- d x y: union(x, y'), union(x', y)

Maintain a size at the root of each group. In union(x, y), compute the size of the new group = x.root.size + y.root.size. Red.size and Black.size are initially zeroes. All cards initially have size 1 (including x'). Whenever Red.size + Black.size = 2n, we report a solution.

Time: O(n + q)

Red, Black Cards (Solution 2)

Maintain a color at each group's root.

- r x: x.root.color ← red, x'.root.color ← black
- b x: x.root.color ← black, x'.root.color ← red
- s x y: union(x, y), union(x', y')
- d x y: union(x, y'), union(x', y)

Maintain a size at the root of each group (same as Solution 1).

Initially every card's color is set to "unknown". In union(x, y), if x.color and y.color are not "unknown" but they differ, then there is a conflict.

Time: O(n + q)

Islands

Solution was described in the lecture slides.

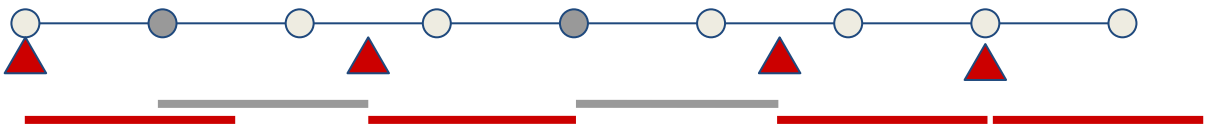
Highlights:

- We must sort the queries by decreasing heights and process them in this order.
- Suppose we just processed h_{prev} and are now processing $h_{now} < h_{prev}$. We shall then find all the cells that have a height in $(h_{now}, h_{prev}]$ (emerging cells).
  - We cannot iterate every cell in the grid to find emerging cells, which takes O(nm) per query and voids the effect of sorting.
  - We should also sort the cells by decreasing height and process them with a queue. Upon h_{now}, we pop the queue until its head <= h_{now}.

Time: O(nm + q) union find + O(nmlog(nm) + qlogq) sorting

There is a road as a segment [0, L] (L<=10^9). There are N (N<=10^5) poles at some integer positions on the road. You can additionally install M poles (M<=10^5) anywhere you like. Then you are to install a camera at each of those N+M poles. A camera on a pole at position x can monitor the road segment [x, x+R], where R is a uniform monitor range for all camera. Your goal is to minimize R by finding the best positions of the additional M poles so that the entire road can be monitored. Determine this minimum R.

Sample: L = 8, N = 2 (at grey points), M = 4



Minimum R = 1.5. Can install the M poles at these red triangles' positions.

- The solution space is monotonic
  - If R works, then R' > R also works.
  - If R doesn't work, then R' < R also doesn't work.
- Given R, we can greedily check if it works.
  - Each of the existing N poles cover some segments.
  - We can merge these segments.
  - Then we can greedily fill the gaps (uncovered segments).
  - For a gap of length X, we need to install ceil(X/R) poles.
  - If the total number of poles we need is greater than M, then R must be increased. Otherwise R can be decreased.

Time: O(logL * N + NlogN). NlogN is the time to sort the existing N poles. Greedily check if an R works takes linear time to merge the N ranges and compute the number of poles needed to fill the gaps.

- For two double variables l and r, while (l <= r) may not terminate if we let m = (l+r)/2 and set l ← m or r ← m.
- It is easier to loop for a constant number of times.

```
double binarySearch(double l, double r) {
  int times = 100; // depends on the precision you want
  while (times--) {
    double m = (l + r) / 2;
    if (condition(m)) r = m;
    else l = m;
  }
  return l; // same as "r" or "(l+r)/2" as they are very close
}
```

Out of Gas

You want to get back home but your car is out of gas. Fortunately your home is at the bottom of a hill that is D distance away ($D<=10^4$). Gravity will pull your car down the hill with a constant acceleration A ($A<=250$).

Unfortunately, there is another car in front of you and you cannot move past it. That car is moving in constant speed down the hill (but its constant speed may change over time, this is given by (ti, xi) pairs, $0<=i<2000$). Fortunately, your brakes are very powerful so that you can reduce your speed by any amount.

What is the earliest time you can get home?

Sample

D = 1000.0
t0 = 0, x0 = 20.5
t1 = 25, x1 = 1000.0

A = 1.0, answer = 44.721 (sqrt(1000.0))
A = 9.81, answer = 25

Solution

- The optimal solution can be to wait for some time, then release the brake and keep full acceleration till you reach home (as long as this will not hit the car ahead).
- We can thus binary search the amount of time to wait. If the time is too small, we will hit the other car.

Proof of Correctness

- If there is an optimal strategy (S1) that gets us home at time T, then we can also have a solution (S2) waits for T - sqrt(2D/A) seconds and accelerates without brakes that also arrives home at T.
- We show that for any point X on the way, S2 must arrive at X no earlier than S1.
  - Suppose at some X, S2 arrives first.
  - Then the speed of S1 at X must be greater than the speed of S2 at X (otherwise it has no chance to arrive home at T)
  - However this is impossible, because S2 has no brakes.

There are N cars ($N<=10^5$) moving at constant speed on a 1D axis. Some on moving right and some are moving left. Given their initial locations and speed (both are integers between $[-10^5, 10^5]$), determine how close will the cars get to each other (the shortest distance between the leftmost and rightmost cars).

Sample

x1 = -10, v1 = 1
x2 = 10, v2 = -1
x3 = 11, v3 = -1

answer = 1

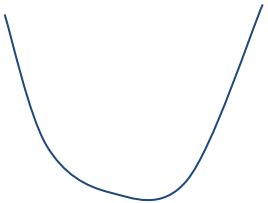
x1 = -10, v1 = 1
x2 = 10, v2 = -1

answer = 0

x1 = -10, v1 = -1
x2 = 0, v2 = 0
x3 = 10, v3 = 1

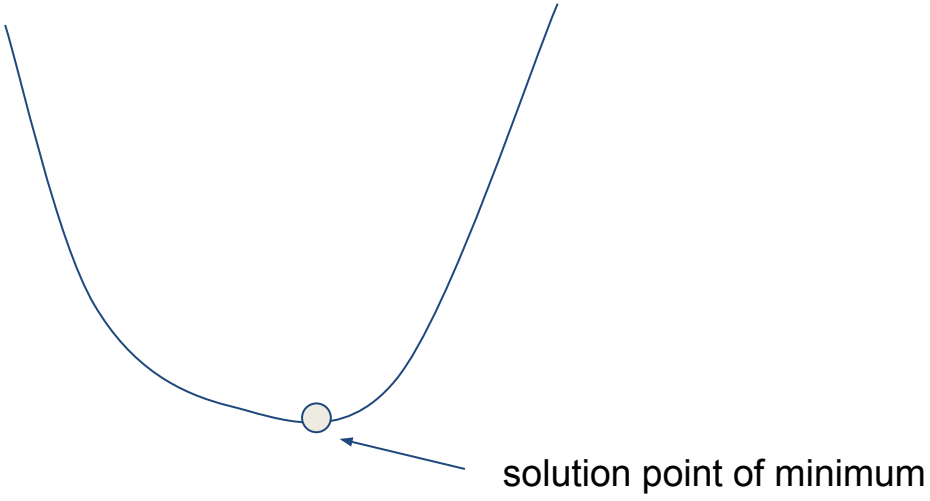
answer = 20

- Consider the distance between the leftmost and rightmost car at a given moment t: D(t).
- D(t) monotonically decreases first and then monotonically increases.
  - In other words, once D(t) increases, it will never decreases later.
  - Proof
    - Consider a moment when D(t) is increasing.
    - Without loss of generality, assume the rightmost car X is going right. The leftmost car Y can either
      - Go left. In this case, cars surpassing X and Y will only make D(t) increase faster.
      - Go right, but slower than X. If D(t) wants to decrease, then there must be another car Z chasing X. But Z must be faster than Y, so that Z cannot be leftmost.

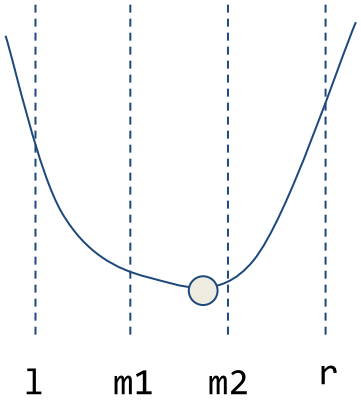


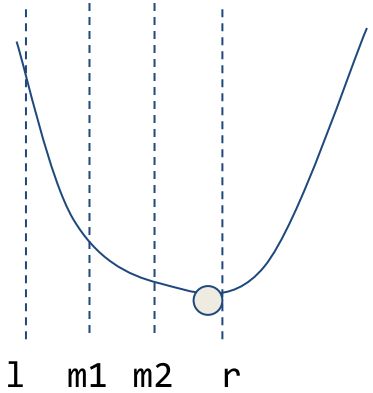
D(t)

- If the solution space is strictly decreasing and then increasing (or vice versa), and we want to find the minimum/maximum point, then we can use ternary search.

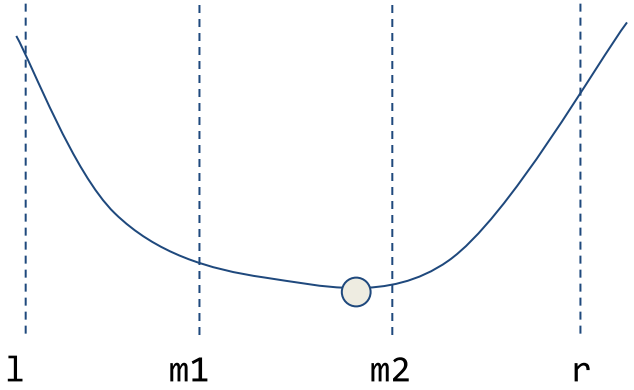


- Cut the search space into three equal parts
  - m1 = l + (r - l) / 3
  - m2 = r - (r - l) / 3
- If f(m1) > f(m2), l = m1; Otherwise r = m2



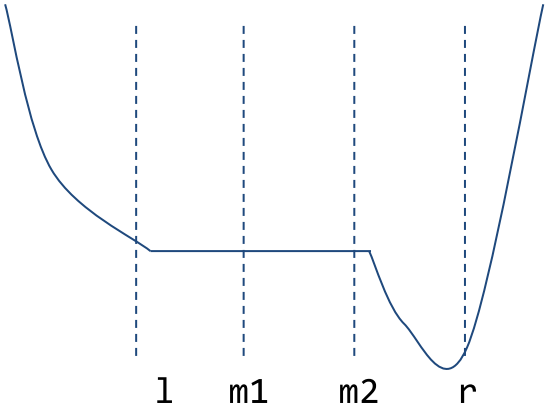


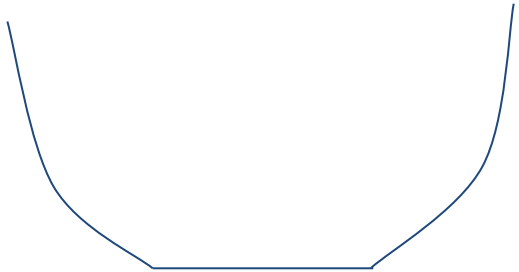
f(m1) > f(m2)



f(m1) > f(m2)

- The solution function must be *strictly* decreasing/increasing.
  - A plateau not at the bottom breaks ternary search.
- A plateau at the bottom is fine.





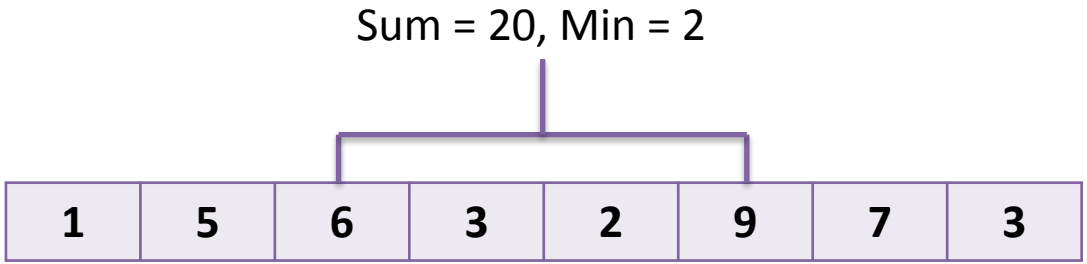
OK, ternary search applies

You have 4 lists of integers A, B, C, D. Each list has at most 1000 integers. Count how many quadruplet (a, b, c, d) there are so that $a + b + c + d = 0$, $a \in A$, $b \in B$, $c \in C$, $d \in D$.

Sample
A = [1, 2, 3]
B = [-1, -2]
C = [0]
D = [-1, -2]
There are three quadruplets (2, -1, 0, -1), (3, -1, 0, -2), (3, -2, 0, -1)

- Let N be the max length of a single list, N<=1000. There are at most $O(N^4)$ quadruplets. We cannot enumerate them.
- We can divide the problem into two parts.
  - We enumerate the pairwise sum of (a, b) $\in$ A × B in $O(N^2)$ time
  - We enumerate the pairwise sum of (c, d) $\in$ C × D in $O(N^2)$ time
  - Then for each sum of (a, b), we just need to look for its negation in the sums of (c, d).
  - We can either use data structures to store and look up those pairwise sums, or we can sort them and apply two pointers. Time is $O(N^2 \log N)$.
- The D&C idea of MITM can also help reduce the search space in complete search.

- A query that asks about a range of elements.
- Examples
  - Range Sum Query
  - Range Minimum/Maximum Queries (RMQ)



- Given an array of integers A[1..n], you are to answer many range sum queries. Each query gives a range [l, r].
- We can create a prefix sum array S[i] = A[1..i]. Then for any range sum query (l, r), we can answer in O(1) time by giving S[r] - S[l-1].

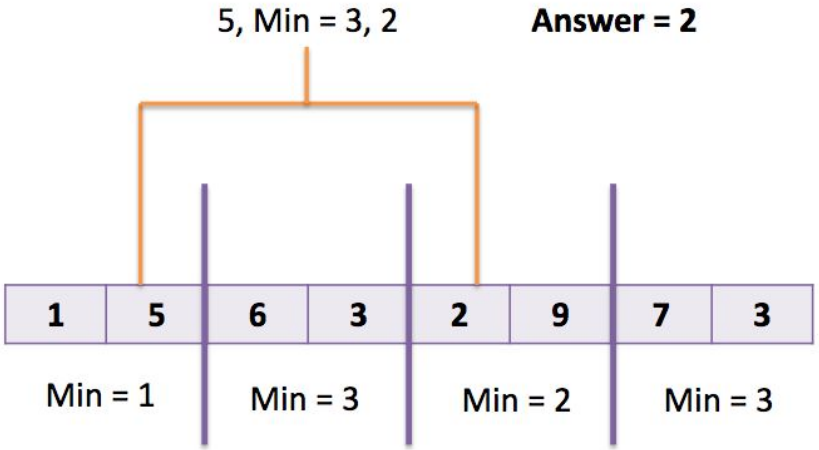
Example
A = [x, 1, 3, 5, 2, 4, 6]
S = [0, 1, 4, 9, 11, 15, 21]

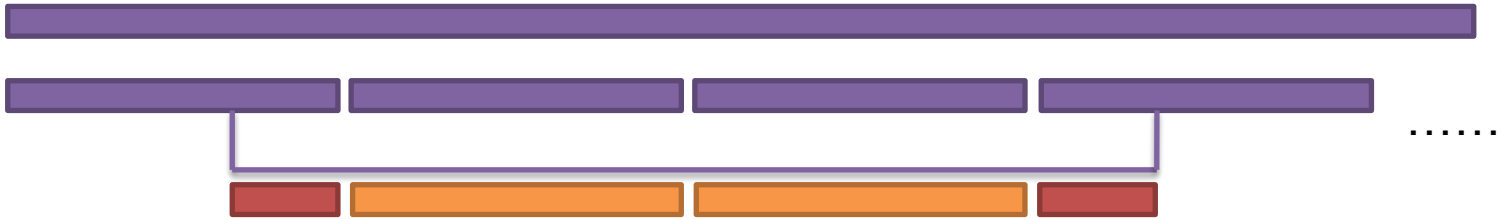
Range sum [2, 5]? => A[2..5] contains 3,5,2,4 => S[5] - S[1] = 15 - 1 = 14
Range sum [1, 3]? => A[1..3] contains 1,3,5 => S[3] - S[0] = 9 - 0 = 9

Prefix Sum (cont'd)

- Prefix sum is a simple way to answer range sum in O(1) time. But it only works on static arrays that do not change.
- Prefix sum does not work for RMQ, because the minimum has no prefix property. That is, we cannot derive min(A[l..r]) given prefix min(A[1..i]) for every i.
  - e.g. For the array A[1..n] = [1, 2, 3, 4, 5], the prefix min is [1, 1, 1, 1, 1], which does not help with answering min(A[3..5])

- The idea of answer RMQ is to apply decompositions.
- In the example below, we divide the array into 4 equal blocks. We precompute the min for each part. The answer to the query is given by checking
  - the precomputed min of one block, and
  - two individual elements.





Block size = B Array length = N

The query range $[l, r]$ has length $L = r - l + 1 \leq N$

Then we have in the worst case:

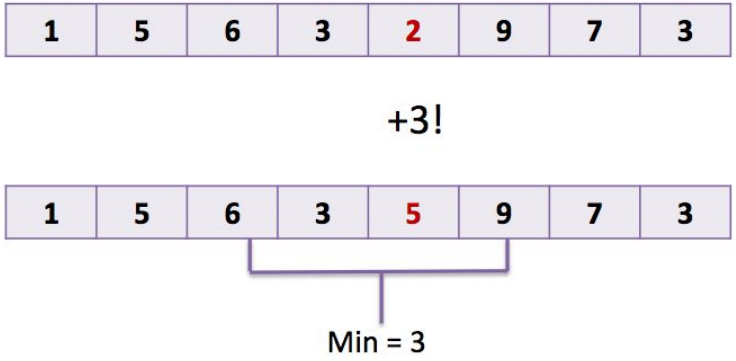
- 2 **red** blocks. Linear cost w.r.t. block size = $2B$.
- L/B **orange** blocks. $L/B \leq N/B$.

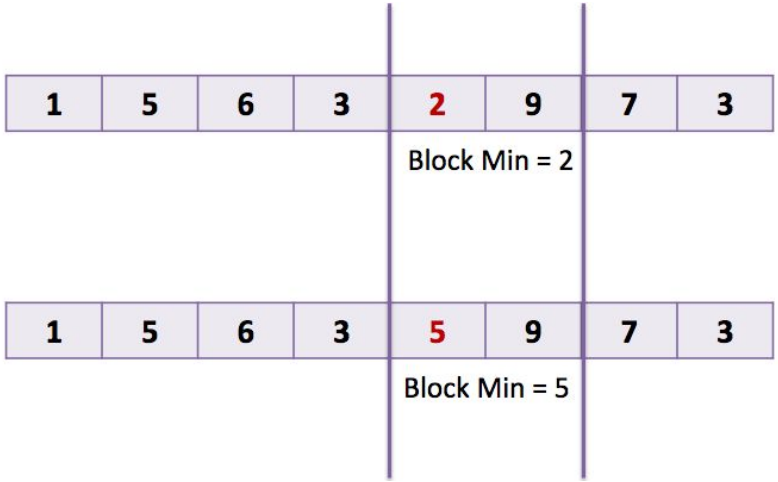
Complexity = $N/B + 2B = O(N/B + B)$.

- As we have query time O(N/B + B), we can choose B = sqrt(N) to achieve the best query time.
- We can store a *maintained value* on each block. In the RMQ case, this value is the minimum of the block.
- For a RMQ query [l, r]:
  - Brute-forcingly check each element for the two blocks at the two endpoints
    - O(sqrt(N)) elements
  - Get the maintained values from the blocks in the middle
    - O(sqrt(N)) blocks

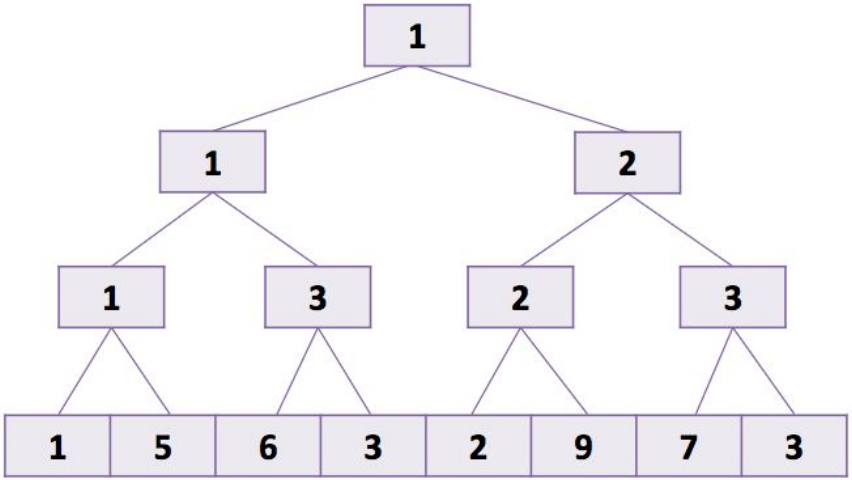
- For Q queries, the total time is O(Q*sqrt(N)). This works for 3-4 sec TL when N, Q<=10^5.

- What if we change a single number in the array?
- We can recompute the min for the affected block in O(B = sqrt(N)) time per update.

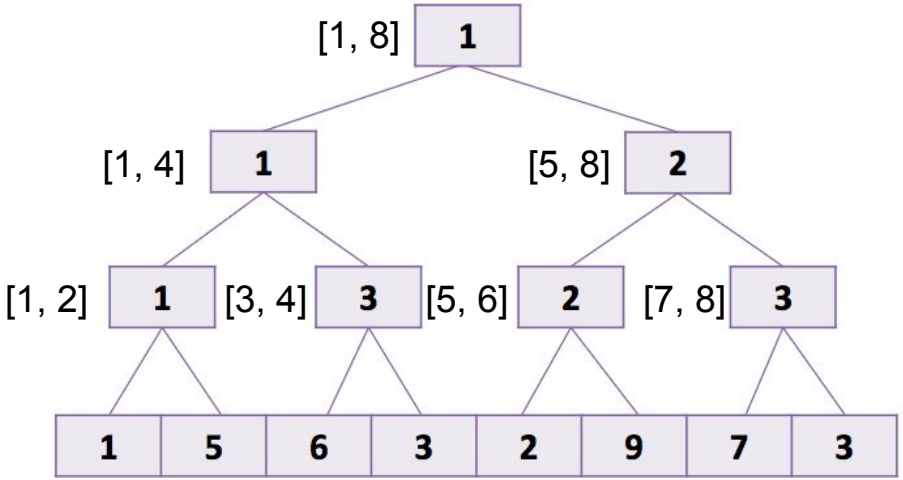




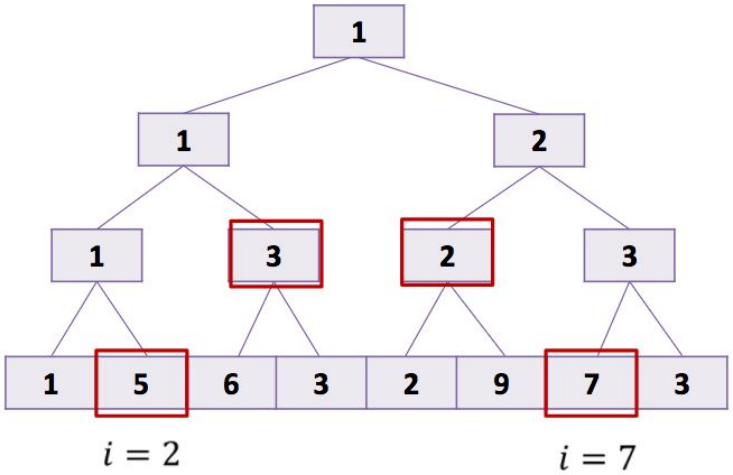
- Square root decomposition is a nice idea. But we can design a more efficient binary decomposition.
- Let's build a binary tree in which each node stores the min of its two children as the maintained value.



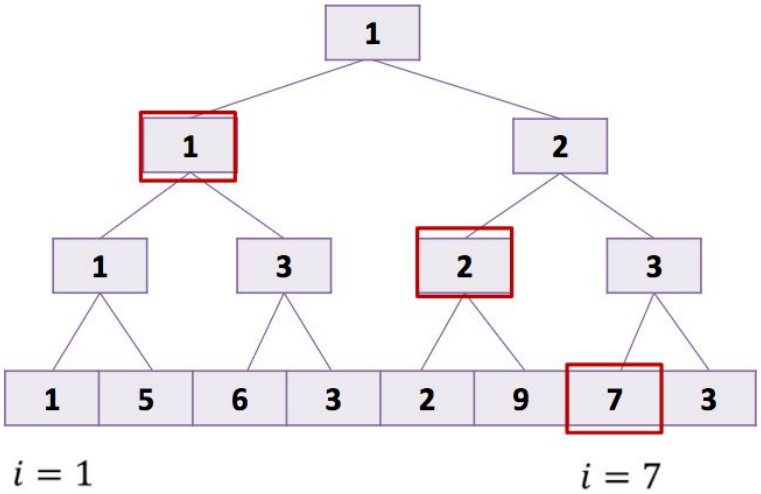
- A popular data structure in APS
- If a node X manages the range [l, r], then its left child manages [l, m], its right child manages [m+1, r], where m = (l+r)/2 (integer division).



- Answering a query [l, r] is to identify a set of *topmost* nodes that cover the range [l, r]. These are the *hit nodes*.
  - Topmost: if both of node X's children can be chosen, we should choose X.
- For any query, the maximum number of hit nodes is O(logn).



Query [2, 7]



Query [1, 7]

- When a node's value is changed, we follow the parent chain and update the min on every node in O(logn) time.