Advanced Data Structures for Olympiad Programmers

Papangkorn Apinyanon

January 31, 2026

"The joy of finding things out is the greatest pleasure there is."

Richard P. Feynman

Contents

Chapter 1

Introduction

The International Olympiad in Informatics (IOI) is one of five international science olympiads. It is essentially a programming competition. Contestants are required to code in C++, solving three problems each day for two days.

The problems often require elegant mathematical intuition, observation, and/or firm knowledge in data structures and algorithms.

The first IOI was held by UNESCO in Bulgaria in 1989. To participate in the IOI, one needs to be selected to represent their country. Each country sends four students who compete individually.

The team selection test (TST) process differs for each country; however, most countries' processes share striking similarity.

1.1 Spirit of OI Competitions

Each problem presents a puzzle: modeling it mathematically, reasoning its structure, and devising an efficient solution.

OI competitions are more than technical skill and knowledge; they test one's ability to think systemically and stay focused under the stressful contest room. A huge part of contests is time management. With half an hour left on the clock, would you try to fix a bug hidden in hundreds of lines of code, or cut losses and move on to work on other problems?

The ability to make such choices defines medalists.

1.2 Prerequisite

This book does not aim to be a basic programming book – readers are expected to have firm knowledge on graph theory; dynamic programming; pointers and memory management; time complexity; standard data structures such as binary heap, balanced binary search tree; and standard algorithms such as Dijkstra's. Essentially, the target audiences are USACO Gold and Platinum level coder – they are assumed to be proficient in using C programming language and to know the basics of C++ standard template library.

1.3 Notation

On a cartesian plane, the main horizontal axis that is the line $y = 0$ is called the OX axis; O means the origin. The main vertical axis that is the line $x = 0$ is called the OY axis. Given a point $P = (a, b)$ on the plane, we say that a, the distance from the point to OY axis, is the *abscissa* of point P, and we say that b, the distance from the point to OX axis, is the *ordinate* of point P.

On a cartesian space, there are axes OX, OY, OZ. The plane containing OX and OY axes we call the OXY plane. The OXZ and OYZ planes are defined similarly. A point $P = (a, b, c)$ on the space has *abscissa* a, *ordinate* b, and *applicate* c. As there is no standard name for the distance to from a point in fourth dimension to the OXYZ space, we call it simply as the fourth coordinate.

When a set of N objects with total order – such as integers between zero and a billion – is given. We can provide a function from the each member of the set into unique integer ID from 1 to N. This process we refer to as *discretization*. Other sources may refer to it as *coordinate compression*.

All logarithms are in base-2 unless otherwise stated.

1.4 Graph Theory

Definition 1. A *graph* $G = (V, E)$ consists of a set of *vertices* V and a set of *edges* E.

A graph is also called an undirected simple graph. We typically denote the order $|V| = N$ and the size $|E| = M$.

Definition 2. Vertices u and v are adjacent in graph G if and only if there exists edge (u, v).

Definition 3. Vertex u and edge e are incident if u is one of the endpoints of e.

Definition 4. A walk is a list of vertices $v_0, v_1, \ldots, v_k$ where each consecutive vertices are adjacent.

Definition 5. A trail is a walk where no edge is repeated.

Definition 6. A path is a trail where no vertex is repeated.

Definition 7. A cycle is a walk where no vertex is repeated, except that the first equal the last.

Definition 8. A tree can be defined by any of the statements following, as they are equivalent; (i) a minimally connected graph. (ii) a graph where its size is one less than its order. (iii) a maximally acyclic graph.

A tree is a minimal connected graph, for deleting any edge cuts it into two. A tree is a maximal acyclic graph, for adding any edge makes a cycle. Given any two different vertices u, v in a tree, there is only one path between the two vertices; we denote this path as u-v path. This property is most important.

Definition 9. A rooted tree T is a tree with a vertex r chosen as root.
We say that u is an ancestor of v if and only if the r-v path contains u. Consequently v is a descendant of u. The depth of a vertex u is the number of edges in r-u path. We say that p is a parent of u if an edge (p, u) exists and p is an ancestor of u. The subtree u is the tree induced by only descendants of u.

Definition 10. The lowest common ancestor (LCA) of u and v in a rooted tree is the vertex with maximum depth which contains both u and v in its subtree.

Definition 11. Tree flattening On a rooted tree where vertices are numbered from 1 to N, running a depth-first-search process starting at the root generates discovery time and exit time of each vertex. The discovery time of a vertex u, denoted by `dfn[u]` or `tin[u]` is one plus the number of vertices visited before it. The exit time of a vertex u, denoted by `tout[u]`, is discovery time of the vertex

added by one less than the number of vertices in its subtree. The discovery time list, `dfn[]`, is a permutation of $1..N$. Its inverse permutation we call `nfd[]`; it satisfies `nfd[dfn[u]] = u` for all vertex u.

Take any vertex u and consider the interval $[dfn[u], tout[u]]$, the vertices which discovery time belongs in the interval is precisely those in subtree u.

Chapter 2

Tricks, Data Structures, and Algorithms

Conquering Standard Problems

A standard task refers to a task that meets one or more of the following criteria: its solution can be easily found online, it lacks novelty, or it consists of well-known techniques that require no significant insight.

Most tasks are unoriginal

For almost every task one may see in contest, there are some existing tasks with close relation to it. The more problems one solves in practice, the more likely they are to notice the relation. For example, IOI24_mosaic closely resembles ARC107E, which predates IOI 2024.

The more advanced tricks and data structures you utilize, the fewer observations you need to make. This chapter introduces you to those.

2.1 Fenwick Tree (Binary Indexed Tree)

Efficient point updates and prefix queries in $\mathcal{O}(\log N)$ time.

A **Fenwick Tree** (also called **Binary Indexed Tree** or **BIT**) is a data structure that provides efficient methods for:

- Point add: add `x` to a single element in an array

- Prefix sum query: compute the sum of the first i elements

Both operations run in $\mathcal{O}(\log N)$ time, where N is the size of the array. The structure requires only $\mathcal{O}(N)$ space.

2.1.1 Intuition

Consider a standard array $A[1..N]$. A naive prefix query requires $\mathcal{O}(k)$ time by summing elements one by one. To improve this, we can precompute prefix sums: $P[i] = A[1] + A[2] + \cdots + A[i]$. This gives $\mathcal{O}(1)$ queries but makes updates $\mathcal{O}(N)$ expensive.
The Fenwick Tree achieves a balance: both operations are $\mathcal{O}(\log N)$ by exploiting the binary representation of indices.

2.1.2 Lowbit Operation

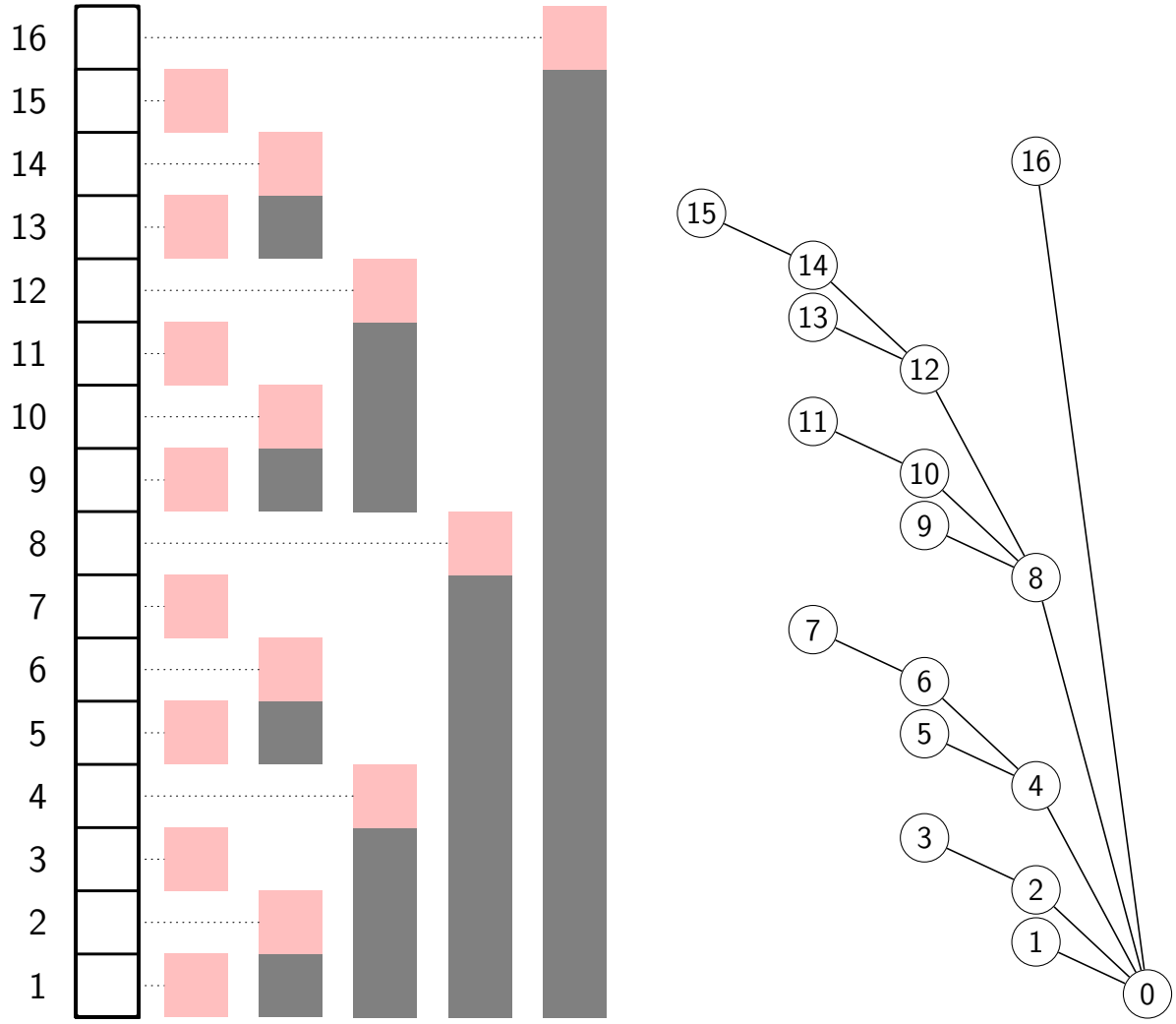
The key operation is `lowbit(i)`, which returns the value of the least significant set bit of i. For example, `lowbit(12)` $=$ `lowbit(`1100_2`)` $= 100_2 = 4$.
To implement `lowbit` in constant operation, we should first understand Two's Complement. In most if not all computers to-day, negative integers are internally stored with Two's complements. Let us consider a hypothetical four-bit signed integer types. 5 would be represented as 0101. What would -5 be?
Since the integer type holds four bits, any overflow is truncated. That is, if we add 0101 to 1011, resulting in 10000, that high bit cannot be stored, and hence the result is 0000. And that is the principle of Two's Complement; any number added to its negative must be zero. Then, one can see that the binary representation of $-x$ is obtained by flipping all the bits in x then adding one to it. Now, the `lowbit` operation is obtained by: lowbit(x) $=$ x & -x.

2.1.3 Structure

A Fenwick Tree stores at each position i the sum of a range of the original array:

$$T[i] = \sum_{j=i-\text{lowbit}(i)+1}^{i} A[j]$$

This forms a hierarchical decomposition. Each index i is responsible for a range of size `lowbit(i)`. See Figure 2.1.



(a) T[] and A[] relationship **(b)** Query pathway (0 = stop)

Fig. 2.1: Fenwick tree for $n = 16$.

2.1.4 Point Update

To add value v to position p:

```
1  void update(int p, int v) {
2    while (p <= N) {
3      T[p] += v;
4      p += lowbit(p);
```

```
    }
  }
```

The algorithm adds `v` to `T[j]` for each j whose responsible range contains j.

Lemma 12. The update algorithm modifies $\mathcal{O}(\log N)$ nodes.

Proof. Each iteration adds `lowbit(p)` to p, flipping the least significant set bit. The number of set bits in N is at most $\lfloor \log_2 N \rfloor$, so the loop runs $\mathcal{O}(\log N)$ times. □

2.1.5 Prefix Query

To compute query$(k) = \sum_{i=1}^{k} A[i]$:

```
int query(int k) {
    int result = 0;
    while (k > 0) {
        result += T[k];
        k -= lowbit(k);
    }
    return result;
}
```

The algorithm climbs down the 'staircase', fetching the sum from those indices along the way. The ranges that belong to those indices are disjoint. Consider, `query(13)` = `query`(1011_2); it visits `T[13]`, `T[9]`, `T[8]`. The ranges $[13, 13]$, $[9, 12]$, and $[1, 8]$ are disjoint and exactly cover $[1, 13]$.

Lemma 13. The query algorithm visits $\mathcal{O}(\log N)$ nodes, and these nodes form a partition of $[1, k]$ into disjoint intervals.

Proof. Each iteration of the loop removes the least significant set bit from k, so the number of iterations equals the number of set bits in k, which is at most $\lfloor \log_2 k \rfloor$.

□

2.1.6 Point Query

Of course, just as one can query [a, b] by [b] - [a - 1], [a, a] is trivial by [a] - [a - 1]. But it is wasteful. One can halves the computation for this special case.

```
int getpoint(int i) {
  int res = T[i];
  int z = i - (i & -i);          // start-1 of T[i]'s covered range
  for (i--; i != z; i -= i & -i) res -= T[i];
  return res;
}
```

2.1.7 Binary Lifting on Fenwick

It is often desirable to find the first index p such that $\sum_{i=1}^{p} A_i >= W$. Of course, performing normal binary search together with querying the tree yields $\mathcal{O}\left(N \lg^2 N\right)$. There exists a $\mathcal{O}\left(N\right)$:

```
int lower_bound_fw(int W) {
  /* return first p with prefix sum >= W; N + 1 if none */
  for (int k = 1 << __lg(N); k; k >>= 1)
    if (p + k <= N && T[p + k] < W) W -= T[p += k];
  return p + 1;
}
```

Problem 14 (Inversion Count)**.** Given an array $A[1..N]$, count the number of pairs (i, j) such that $i < j$ and $A[i] > A[j]$.

Solution

First, discretize the array so that elements are in range $1 \ldots N$. Process elements from right to left. For each position i, query the BIT for the number of elements smaller than $A[i]$ seen so far. Then, add $A[i]$ to the BIT.

```
long long count_inversions(int A[], int N) {
  Discretize(A, N);
  memset(T, 0, sizeof T);
    long long ans = 0;
    for (int i = N; i >= 1; i--) {
```

```
        ans += query(A[i] - 1);
        update(A[i], 1);
    }
    return ans;
}
```

Time complexity: $\mathcal{O}(N \log N)$.

2.2 Segment Tree

2.3 Sprase segment tree

2.4 Copy-on-write segment tree

2.5 Fenwick Tree for Maximum (and Why 2D BIT Fails)

1D BIT works for max, but 2D BIT fails for non-commutative operations.

The 1D Fenwick Tree can be adapted to support maximum instead of sum. The key insight is that for additive operations like sum, disjoint interval decomposition works perfectly. However, this breaks down in 2D for non-commutative operations like max or min.

2.5.1 Fenwick for Maximum

The one dimensional Fenwick tree can be adapted to support prefix maximum instead of sum. This is because max of the disjoint intervals is the max of their union.

```
int query_max(int k) {
    int result = -INF;  // or appropriate minimum value
    while (k > 0) {
        result = max(result, T2[k]);
        k -= lowbit(k);
```

```
    }
    return result;
}

void update_max(int p, int v) {
    while (p <= N) {
        T2[p] = max(T2[p], v);
        p += lowbit(p);
    }
}
```

Now, it is noteworthy that Fenwicks can be modified to support suffix-sum query instead of prefix-sum. This is done by having the update function climb downstairs and the query function go up, swapping their paths. Alternatively, one may wrap the functions with $p = N + 1 - p_0$.

2.5.2 Why 2D BIT for Maximum Fails

A 2D Fenwick Tree is conceptually: a 1D BIT over the x-dimension, where each x-node contains a 1D BIT over the y-dimension.
When querying a prefix $[1, x] \times [1, y]$ (a rectangle from the origin):
For sum (additive, commutative): The BIT decomposition gives us rectangles that may overlap. Since addition is commutative and associative, overlap doesn't matter:

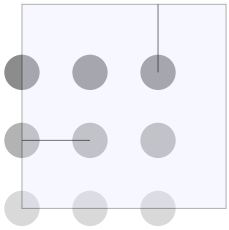
$$\text{sum}(A + B) = \text{sum}(A) + \text{sum}(B)$$

For max (non-commutative, structure-dependent): The same decomposition creates overlapping rectangles. This causes two problems:

1. **Missing points**: Some points in the query rectangle may not be covered by any BIT rectangle
2. **Irregular coverage**: The "remaining" parts after subtracting BIT rectangles don't form nice axis-aligned sub-rectangles

Lemma 15. For non-commutative non-associative operations, a query decomposition must be a true partition of the query region into disjoint sub-regions.

2D BIT cannot guarantee this for arbitrary rectangles defined by $(1, x)$ and $(1, y)$ corners.



BIT rectangles overlap in 2D

Fig. 2.2: 2D BIT creates overlapping rectangles

Lemma 16. For non-commutative non-associative operations, a query decomposition must be a true partition of the query region into disjoint sub-regions.

2D BIT cannot guarantee this for arbitrary rectangles defined by $(1, x)$ and $(1, y)$ corners.

2.5.3 Working Alternatives for 2D Max Queries

- **2D Segment Tree**: $\mathcal{O}(\log^2 N)$ query, $\mathcal{O}(\log^2 N)$ update. Recursively traverse the x-tree, and for each relevant node, query the y-BIT for maximum.
- **Sqrt Decomposition**: Each x-node stores a BIT over y. For large ranges, combine precomputed values.
- **Segment Tree of Segment Trees**: A segment tree over x where each node is a segment tree over y.

Structure	Query Time	Update Time
2D BIT (sum)	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log^2 N)$
2D BIT (max)	Fails	Fails
2D Segment Tree	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log^2 N)$
Sqrt Decomposition	$\mathcal{O}(\sqrt{N} \log N)$	$\mathcal{O}(\log N)$

Table 2.1: 2D range query data structures

2.6 Merge small to large

Problem 17 (CSES Distinct Color)**.** On a tree rooted at node 1 where each node u has its color C_u, find for each subtree the number of distinct colors in it.

Solution

Run a depth-first-search, for each node u we maintain an instance of `std::set` named S_u. At the moment we leave a node u, S_u shall contain all the colors in its subtree.

We say that a child v of u is heavy if and only if the size of subtree v is at least half the size of subtree u. A child v of u is light if it is not heavy. A node either have zero or one heavy child. On our `DFS(u)` process, once we traversed all of its children we do the following to adjust S_u to its desired state: If u has a heavy child h, inherit its set – don't copy the set over, but change the pointer S_u to the set S_h directly – this can be done by `S[u].swap(S[h]);` in C++ – this operation takes constant time. Now for each light child v of u, we iterate over everything in S_v and insert them into S_u in a brute-force manner. Lastly, we insert C_u to S_u, and note the size of S_u into $answer_u$.

Lemma 18. This solution takes $O(N \lg^2 N)$.

Proof. The color element generated by each vertex u only gets moved up $O(\lg N)$ time, for each time we move from a light child to its parent the size of subtree in focus increases twofold. Hence, we do $O(N \lg N)$ moves in total. An extra log factor is added by the `std::set`. □

Problem 19 (CF600E – Lomsat Gelral)**.** Given a tree rooted at 1, with each vertex u is a color c_u. We says that a color c dominates subtree u if no color appears has more occurance than c does in that subtree. For each subtree, print the sum of all of its dominating colors.

Solution Use a `std::set` of `(frequency, color)` pair. The set, being a binary search tree, can fetch the highest frequency. Maintain the sum of dominating colors in addition to it. Then apply merge small to large technique.

2.7 DSU on tree – Sack

This trick is very closely related to *Merge small to large*, and some people use the terms interchangeably; however, there is a striking difference between them: *We need one instance of data structure in Sack, instead of the N instances in Merge small to large*. The general theme is the same –

> *For each subtree, what would the state of a data structure D be if only the vertices in the subtree is in the data structure?*

Problem 20 (CSES Distinct Color)**.** On a tree rooted at node 1 where each node u has its color C_u, find for each subtree the number of distinct colors in it.

Solution First, we assume the colors are integers between 1 to N, discretize them if they are not. We can use a simple array to support these three operations in constant time: add an instance of a color, remove an instance of a color, and count distinct colors. We were forced to use `std::set` on the Merge small to large solution, for the array would take $O(N)$ memory and we cannot have N instances of it.

However, here is a trick that let us use one instance. We define heavy and light child the same way. Run a depth-first-search `DFS(u, keep)` where `keep` is a boolean. When leaving node u, the data structure should have already seen the state where only the vertices in subtree u is put in. If `keep` is true, it shall stays in that state as we leave the node; if `keep` is false, we shall clear and empty the data structure before we leave the node.

After entering node u in the depth-first-search – D is at that time empty – run `DFS(v, false)` for each of its light child v. D is still empty when we are done. Then, run `DFS(h, true)` if u has a heavy child h. The data structure now contains vertices in subtree h. We now iterate over all vertices in subtree u that is not in subtree h and insert them into D. Precisely at this point we visit the state of D when the vertices in subtree u are exactly those in D – take note of the answer we want. Then if `keep` is false in the current depth-first-search call, empty the data structure by removing everything.

Directly performing this trick and using the array as D solves this problem in

$O(N \lg N)$.

Lemma 21. This process performs $O(N \lg N)$ `add` and `remove` operations on D.

Proof. Fix a vertex x. Consider the unique path from x to the root.

A single vertex x is (re)inserted into D exactly when x lies in a light subtree of some ancestor u while u is being processed. Each time this happens, the size of the currently "kept" part of the tree (the heavy child's subtree) at that ancestor is at least twice the size of the light subtree containing x. Therefore, moving up along the ancestors where x belongs to a light child, the size of the context into which we merge at least doubles each time. The doubling can occur at most $\lfloor \lg N \rfloor + 1$ times.

Hence, each vertex participates in $O(\lg N)$ `add` (and matching `remove` for the temporary light traversals) operations. Summed over all N vertices, the total number of operations on D is $O(N \lg N)$. □

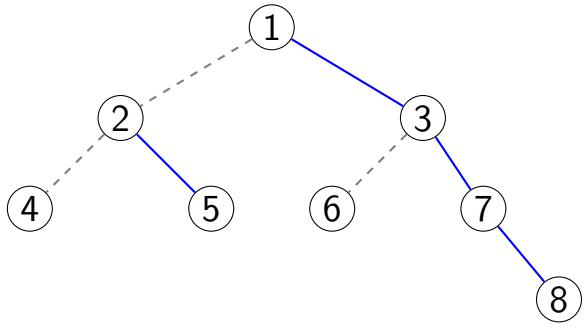
Problem 22. All problems solvable with Merge small to large are solvable with Sack. Refer to previous section for more practice problems.

2.8 Heavy-Light Decomposition

Heavy-Light Decomposition (HLD) is a technique for breaking a tree into disjoint paths; so that any path query can be expressed as a small number of contiguous segments in an array. It reduces path query problems to array range query one.

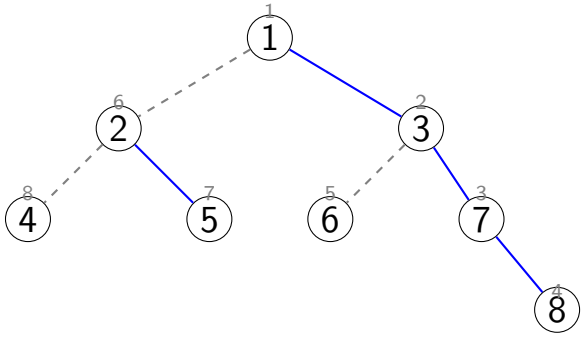
Motivation. In many problems we must process queries along paths between two verticessuch as maximum edge weights. Traversing a path directly is too slow for large N. HLD helps us look consider only $O(\lg N)$ segments.

Concept. For each node, mark the edge to its largest subtree child as *heavy*; all others are *light*. Heavy edges form chains that represent dense parts of the tree; all chains end at a leaf node. Any root-to-leaf path crosses at most $\lg N$ light edges, for each move from a vertex to its light children halves the size of the active subtree.



Heavy edges: solid blue
Light edges: dashed gray

Building the decomposition. Compute subtree sizes by DFS. For each vertex, select its child with the largest subtree as heavy. Nodes connected by heavy edges form a *heavy path*; each path has a *head*. Linearize the tree by traversing heavy paths consecutivelythis allows segment tree indices to correspond to vertex order.



DFS order: heavy-first numbering

Path queries. To query the path (u, v), repeatedly lift the deeper vertex to the head of its chain, querying each contiguous segment until both vertices lie in the same chain, where a final range query covers the remainder. Each lift crosses one light edge, hence at most $2 \lg N$ iterations.

```
vector<int> g[N];
int sz[N];

void dfs(int u, int p) {
  depth[u] = depth[p] + 1;
  par[u] = p;
  sz[u] = 1;
  for (auto &v: g[u]) if (v != p) {
    dfs(v, u);
    sz[u] += sz[v];
    if (g[u][0] == p or sz[v] > sz[g[u][0]])
      swap(g[u][0], v);
  }
}

void dfs2(int u, int p, int head_) {
  hld[u] = head_;
  dfn[u] = timer++;
  for (auto &v: g[u]) if (v != p)
  dfs2(v, u);
  out[u] = timer;
}
```

You can also find LCA by the same logic as path query. This is often faster than the binary lift based method.

```
int lca(int u, int v) {
  while (hld[u] != hld[v]) {
    if (depth[hld[v]] < depth[hld[u]]) swap(u, v);
    v = par[hld[v]];
  }
  return depth[u] > depth[v]? u: v;
}

int path_min(int u, int v) {
  int ans = INT_MAX;
  while (hld[u] != hld[v]) {
    if (depth[hld[v]] < depth[hld[u]]) swap(u, v);
```

```
      ans = min(ans, /* min of nodes with dfs order
            in range [ dfn[hld[u]], dfn[u] ]
            ; can be done with segment tree */);
      v = par[hld[v]];
    }
    ans = min(ans, /* min of nodes with dfs order between
            dfn[u] and dfn[v] */);
    return ans;
  }
```

K-th ancestor.

To find K^{th} ancestor, just jump repeatedly until the answer lies in current chain.

```
int kth(int u, int k) {
  if (k >= depth[u]) return -1;
  while(1) {
    if (depth[u] - depth[hld[u]] >= k)
      return nfd[dfn[u] - k];
    k -= depth[u] - depth[hld[u]] - 1;
    u = par[hld[u]];
  }
}
```

2.9 Centroid Decomposition

Reduce a problem about all paths in the tree to only considering the paths passing through a root.

Definition 23. Given a tree of order N, a vertex u is a *centroid* if, when u and its incident edges are removed, none of the resulting trees exceeds $N/2$ nodes. A tree has one or two centroids.

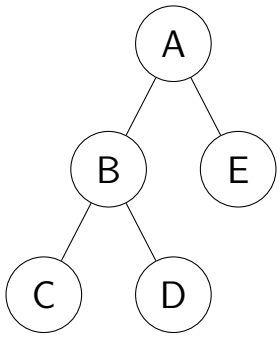


Fig. 2.3: Example tree – B is its unique centroid.

Finding a Centroid

To find a centroid of tree T, choose arbitrary vertex a. Run a depth-first-search rooted at a to find subtree size of each vertex. Then repeatedly do following logic: start at a, look at children of current vertex, if any of them have subtree size exceeding half the order of original tree, move to said vertex; else, the current vertex is a centroid.

Centroid Decomposition Algorithm

Given a tree T, we want to find its centroid tree $T+$. To do that, find any centroid u of the tree, this will be the root of $T+$. Remove u from T, and for each remaining component, recursively decompose them. Attach the root of resulting trees of their decomposition into u.

```
vector<int> g[N], centree[N];
int sz[N], dead[N];
void dfs(int u, int p) {
  sz[u] = 1;
  for (auto v: g[u]) if (!dead[v] && v != p)
    dfs(v, u),
  sz[u] += sz[v];
}
int findcen(int u, int p, int treesz) {
  for (auto v: g[u]) if (!dead[v] && v != p && sz[v] * 2 > treesz)
    return findcen(v, u, treesz);
  return u;
```

```
13      }
14      int decom(int u) {
15        dfs(u, -1);
16        u = findcen(u, -1, sz[u]);
17        dead[u] = 1;
18
19        /* we can do a lot of things here ! */
20
21        for (auto v: g[u]) if (! dead[v])
22          centree[u].push_back(decom(v));
23        return u;
24      }
25
```

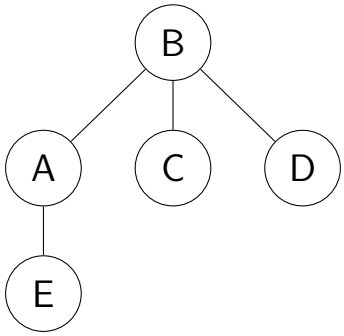


Fig. 2.4: Centroid tree of the example.

Key Properties of Centroid Tree

Lemma 24. The recursion depth of DECOM() – which is the height of centroid tree – is $O(\lg N)$.

Proof. Each recursive call halves the order of focused subgraph. □

Corollary 2.9.1. Each vertex in centroid tree has $O(\lg N)$ ancestors.

Lemma 25. Sum of subtree size over all vertices is $O(N \lg N)$.

Proof. A vertex contributes one to size of subtree of each of its ancestors; each vertex contributes at most $\lg N$ to the sum. □

The following property is extremely useful when dealing with path information tasks:
Let T be a tree and T' be its centroid tree. Then for all uv paths in T, there exists a unique node $\delta(u, v)$ such that the union of $(u, \delta(u, v))$ and $(v, \delta(u, v))$ paths in T is precisely the uv path, and u, v belong to subtrees of different children of $\delta(u, v)$ in T'. That vertex is precisely the lowest common ancestor of u and v in T'.

This means each of the $O(N^2)$ possible simple paths is composed of two among the $O(N \lg N)$ paths from vertices to their ancestors in the centroid tree.

Applications

Problem 26. IOI11_race. Given a tree with weighted edges, find a path such that the sum of its edge weights is exactly L. If multiple paths satisfy this, choose one with the fewest number of edges.

Solution. Let the given tree be T and its centroid tree be $T+$. For each level of decomposition, after we get the centroid c, we will consider all path (u, v) satisfying $\delta(u, v) = c$.

1. For each subtree C_i of c (i.e., components after removing w), perform a DFS to collect all distances from c to nodes in C_i.

2. Maintain an associative array (e.g. `std::map`) that records the minimum number of edges required to achieve a given path length from w.

3. For every node x in the current subtree C_i, check if there exists a distance d' in the map such that $d + d' = L$, where d is the distance from x to w. If such d' exists, update the answer with the sum of corresponding edge counts.

4. After processing C_i, add all its distances and edge counts into the map so that subsequent subtrees can use them.

5. Clear the map when finishing all subtrees of w, and recursively process each C_i.

Assuming L is sufficiently small that we can use an array instead of `std::map`, Since each level processes disjoint nodes and each node appears in $O(\lg N)$ decompositions, the overall complexity is $O(N \lg N)$ efficient enough for $N \leq 2 \times 10^5$.

2.10 Edge Centroid Decomposition

Like point centroid, but cooler!

In **standard centroid decomposition**, we recursively remove a vertex whose removal splits the tree into components of size at most half of the original. In **edge centroid decomposition**, however, we remove an *edge* instead of a vertex.

Definition 27. An edge (u, v) in a tree T of order N is called an *edge centroid* if, when removed, the sizes of the two resulting subtrees differ as little as possible; equivalently, it minimizes the size of the larger component.

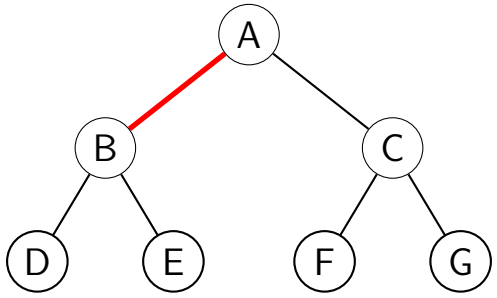


Fig. 2.5: Illustration of edge centroid decomposition: the red edge (A, B) is the centroid edge that splits the tree into two balanced (as much as viable) parts.

This decomposition defines a recursive structure similar to the centroid tree, but using edges as separators instead of vertices.

Bound on Decomposition Depth

In vertex-based centroid decomposition, every recursive call processes a tree of size at most $N/2$, ensuring logarithmic recursion depth $O(\lg N)$.
However, this bound does not hold for edge decomposition. Consider a star graph of order N removing any edge separates only one leaf from the rest, reducing

the size by 1. Thus, the recursion depth in worst case is $O(N)$.

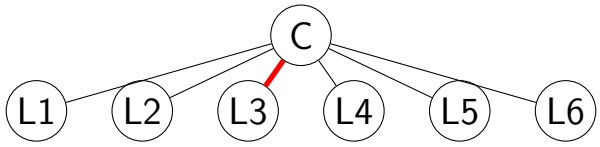


Fig. 2.6: Star graph

Binarizing the Tree

To ensure balanced splitting, we can *binarize* the tree i.e. transform every node into a binary structure by inserting auxiliary vertices. If every node in the resulting tree has degree at most 3, it can be proven that there exists an edge (u, v) whose removal results in two subtrees, each with size not exceeding $\frac{2}{3}N$.
For a formal statement, look up the *Edge-weight Tree Separator Lemma*.
Hence, on a binarized tree, the recursion depth is bounded by:

$$O(\log_{1.5} N) = O(\lg N)$$

This ensures edge centroid decomposition achieves logarithmic height similar to vertex-based decomposition.

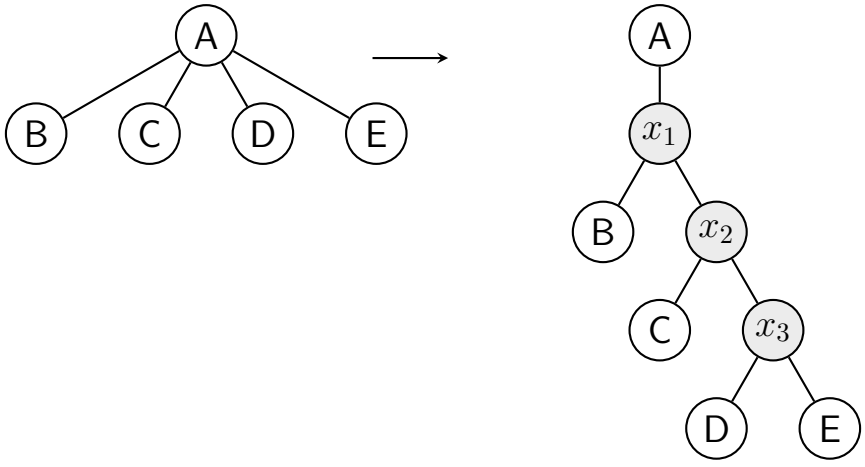


Fig. 2.7: Binarizing the tree: replacing a high-degree vertex by auxiliary nodes (gray) to ensure degree ≤ 3.

Avoiding Dynamic Data Structures

Let us revisit `IOI11_race`, discussed in the previous section.
In vertex-centroid decomposition, we often use associative arrays (i.e., std::map) to store the frequency of distances from the centroid. In the edge-centroid approach, let (x, y) be the current edge centroid; we have only two endpoints. Thus, we can compute distances from each endpoint separately and use sorted arrays for efficient lookup.

1. Compute all distances from x and y to nodes in their subtree to arrays A and B respectively. As the arrays shall be sorted, use breadth-first-search to compute the distances.

2. Perform monotonic queue trick to find pairs (a, b) such that $a + b = L$.

As the subproblem where we only care about paths passing through the root (i.e. the centroid) is now solved in $O(N)$, the original problem can be solved in $O(N \lg N)$.

Edge Centroid Tree

In standard centroid decomposition, a vertex in the original tree share label with the corresponding vertex in centroid tree; it works because the centroids are vertices. As centroids are edges here, it is less straightforward to build a centroid tree.
Say tree T has edge $e = (x, y)$ as its centroid. If e is removed, T will split into T_x, T_y containing x, y respectively. We construct edge centroid tree $T*$ such that e will have a corresponding vertex in the $T*$. Let's denote that node as $\delta(e)$, which is also the root of $T*$. We store following informations in $\delta(e)$: `x, y, leftchild, rightchild`. In addition, of course, any data we wish to store to solve the problem – as done in standard centroid decomposition. `leftchild`, `rightchild` store the root of $Tx*$ and $Ty*$ respectively. If any of Tx or Ty is a singleton tree, then its corresponding root in $Tx*$ or $Ty*$ need not store any data – it acts as a dummy vertex.

An advantage of edge centroid tree over standard centroid tree is that each non-dummy vertex has exactly two children. This means we can do *pushup* operation easily.

Problem 28. CF150E asks for a simple path with the maximal median edge weight in a weighted tree.

Solution. Perform a binary search over the median value. For threshold x, assign each edge weight:

$$w'(e) = \begin{cases} +1, & \text{if } w(e) \geq x, \\ -1, & \text{otherwise.} \end{cases}$$

Now the task reduces to finding whether there exists a path with nonnegative total $w'(e)$.

Using edge centroid decomposition:

- Let (x, y) be the current centroid edge.
- Collect distances from xs side into $A[]$, and from ys side into $B[]$.
- Instead of sorting, use BFS to fill A and B in nondecreasing order of depth.
- Apply a monotonic queue trick to merge A and B, avoiding the $\lg N$ factor from segment trees.

This yields an $O(N \lg^2 N)$ algorithm with low constants.

2.11 Cartesian Tree

A cartesian tree of array $A[1..N]$, denoted by $C(A)$ is defined as follows: if A is empty, $C(A)$ is the empty tree; else, let i be any $argmax(A)$, we create a vertex labeled i and make it the root of $C(A)$. Then, we assign $C(A_1)$ as the left child of i and $C(A_2)$ when A_1 is the prefix of A ending right before i and A_2 is the suffix of A starting right after i.

Some people use the term *cartesian tree* to mean Treap. In this book the term *cartesian tree* exclusively belongs to what is described in this section and the term *Treap* strictly means the balanced binary search tree.

i	1	2	3	4	5
$A[i]$	3	2	5	1	4

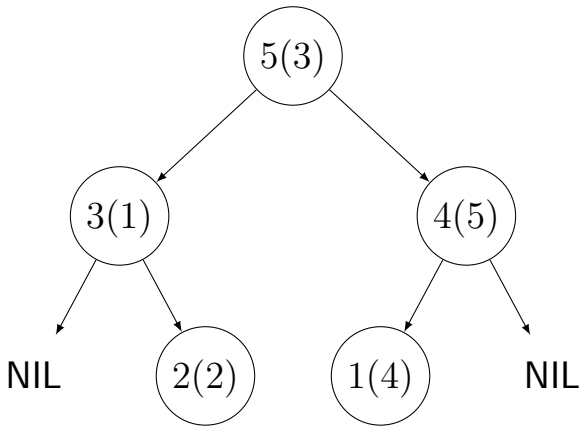


Fig. 2.8: Cartesian tree $C(A)$ for $A = [3, 2, 5, 1, 4]$ In parentheses are the corresponding indices in A.

Building a Cartesian Tree in $O(N \lg N)$

Build any data structure which can answer range maximum query over A – a segment tree works. Then directly implements the recursive building process from the definition of cartesian tree.

Building a Cartesian Tree in $O(N)$

Iterate over the array from left to right, maintaining a stack of the right chain of $C(A)$. As we consider $A[i]$, pop the bottom of the right chain so long as it is less than $A[i]$, for $A[i]$ shall not have it as a ancestor. Then insert $A[i]$ into the right chain, linking up pointers as needed.

```
1 int stk[N], top, k, i,
2   rs[N], ls[N]; /* right and left child respectively */
3 for (int i = 1; i <= n; ++i) {
4   k = top;
5   while (k > 0 && A[stk[k]] < A[i]) --k;
6   if (k) rs[stk[k]] = i;
7   if (k < top) ls[i] = stk[k + 1];
```

```
  stk[top = ++k] = i;
}
/* the root is at stk[1] */
```

Problem 29 (CEOI20_Fancyfence)**.**

2.12 Persistent Segment Tree (copy-on-write)

In a segment tree of size N, we only touches $O(\lg N)$ nodes in `update` function. $\lg N$ is a small number – we have the memory to afford making new nodes instead of editing the existing ones (in most case, at least).

Let us do exactly that, the `update` function take in one root of segment tree, then return a new root, of new version of that segment tree.

We maintain an array `root[0..V]` of pointers to the root for each version, this way we can actually access each version. Version 0 is usually the empty tree.

```
struct Node { int lc, rc; long long sum; } t[MAXM];
int root[MAXV], T;  // T = nodes allocated

int update(int v, int l, int r, int pos, int delta){
  int w = ++T;              // new node
  t[w] = t[v];              // copy fields
  t[w].sum += delta;        // this segment contains pos
  if (l == r) return w;
  int m = (l + r) >> 1;
  if (pos <= m)
    t[w].lc = update(t[v].lc, l, m, pos, delta);
  else
    t[w].rc = update(t[v].rc, m+1, r, pos, delta);
  return w;
}
```

The `update` function here still takes $O(\lg N)$ time as in standard segment tree, but each call takes $O(\lg N)$ space. This has tremendous applications, some are described below.

2.12.1 Rectangle Sum

Given K points on cartesian plane, each one with integer weight. You have Q queries to answer; for i^{th} query, print sum of weight of points lying in the rectangle with bottom-left coordinate at (a_i, b_i) and top-left coordinate at (c_i, d_i).
Persistent segtree solves this problem in $O(N \lg N)$ time and space complexity. First we sort the points by their abscissa such that $i < j \implies x_i \leq x_j$. Create persistent segment tree representing range sum of weights over the ordinates. The first version is an empty tree. The $i^{th}; i > 1$ version has the contribution of weight of point j for all $j \leq i$ at the ordinate y_j. To answer a query, find two versions (E, F) such that E is the last version not including any points with abscissa at least a_i and F is the last version not including any points with abscissa more than c_i. The answer for i^{th} query then is $T_F.query(b_i, d_i) - T_E.query(b_i, d_i)$. As the difference between the two versions is exactly the segment tree we would get if we only add points having abscissa between $[a_i, c_i]$, the abscissa condition is eliminated by the versioning mechanism, and the ordinate condition is handled by the segment tree range query itself.

2.12.2 K-th Order Statistic

Problem 30 (SPOJ MKTHNUM)**.** Given array of integers $A[]$ of length N. Answer Q queries – for i^{th}, print the k_i^{th} least element from the subarray $A[l_i..r_i]$.

First, consider the subproblem where $l_i = 1$. Construct a persistent segment trees where version i is version $i - 1$ added by 1 at $A[i]$. We can take the tree at version r_i and do walk-on-segtree to find first z such that sum from $(-\infty, z]$ on the segment treeis not less than k_i.
Now to solve the full problem, we can think of an imaginary segment tree, where the sum attached to each node is exactly the difference of the corresponding nodes in versions r_i and $l_i - 1$. That is the segment tree we would get if we only add the elements from $[l_i, r_i]$. Now just walk on that segment tree. Since the segment tree is not real, we have to modify our walk function to take in two pointers. Time complexity is $(N \lg N + Q \lg N)$. Space complexity is $(N \lg N)$.
Given K points on cartesian plane, each one with integer weight. You have Q

queries to answer; for i^{th} query, print sum of weight of points lying in the rectangle with bottom-left coordinate at (a_i, b_i) and top-left coordinate at (c_i, d_i).
Persistent segtree solves this problem in $O(N \lg N)$ time and space complexity. First we sort the points by their abscissa such that $i < j \implies x_i \leq x_j$. Create persistent segment tree representing range sum of weights over the ordinates. The first version is an empty tree. The $i^{th}; i > 1$ version has the contribution of weight of point j for all $j \leq i$ at the ordinate y_j. To answer a query, find two versions (E, F) such that E is the last version not including any points with abscissa at least a_i and F is the last version not including any points with abscissa more than c_i. The answer for i^{th} query then is $T_F.query(b_i, d_i) - T_E.query(b_i, d_i)$. As the difference between the two versions is exactly the segment tree we would get if we only add points having abscissa between $[a_i, c_i]$, the abscissa condition is eliminated by the versioning mechanism, and the ordinate condition is handled by the segment tree range query itself.

2.12.3 Tree Path Range Sum

Problem 31 (SPOJ COT)**.** Given a weighted tree of order N with Q queries – for each query print k_i^{th} minimum weight on the simple path from u_i to v_i.

Solution Similar to `MKTHNUM`, make a persistent segment tree where each vertex in a tree has its own version, derived from that of its parent and modified by adding 1 to the position that is the weight of edge above it. After building it, notice that the segment tree we would get if we only add in the edges in u-v path is exactly the tree we would get by summing up these three version: $T_u + T_v - 2 \cdot T_{lca}$. Walk on segtree again, but this time with three parameters.
Extra How to solve this problem if weights were on the vertices?

Problem 32. Given a tree with N vertices, each one having color between 1 and N, and a weight between 1 to 10^9. Answer Q queries – for i^{th} query print the sum of weight of all vertices which lie on the simple path from u_i to v_i and which have color in between x_i and y_i.

Solution This is easier than the previous one, for you do not need to walk on segment tree. Just query the sum and add them up!

2.13 Square Root Decomposition: Blocking

Square Root Decomposition (or sqrt-decomposition) is a classical technique for achieving sublinear query time on static data. It divides the array into *blocks* of size roughly $\sqrt{N}$ and precomputes aggregated information within each block.

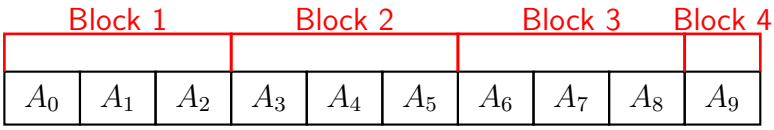


Fig. 2.9: Blocking

Given an array $A[1..N]$, we partition it into blocks of size $B \approx \sqrt{N}$. For each block i, store a precomputed value, like the minimum of values in the block.
To get minimum over range $[l, r]$: First, find blocks lying completely inside the range and consider their precomputed values. Next, consider all remaining elements in a bruteforce manner; there are at most $2B$ such elements. This yields $O(B + N/B)$ time per query, minimized when $B = \sqrt{N}$.

2.14 Square Root Decomposition: Batching

Batching refers to handling multiple operations at once using square root decomposition grouping updates or queries into *batches* of size about $\sqrt{N}$, so that each batch can be processed efficiently as a whole.
Whereas blocking divides a data structure *spatially*, batching divides it *temporally*. Suppose we have Q queries that modify or read the same array. Instead of updating the array after every single query (which can be slow), we split the Q operations into $\sqrt{Q}$ batches. Apply all updates within a batch together, and *buffer* new updates.

Problem 33. We can maintain an array $A[1..N]$ with Q operations:

- `add(l, r, x)` add x to every element in range $[l, r]$,
- `get(i)` query value at position i.

Solution

To solve this, we maintain `buffer` – a list of pending updates. When a new `add` command comes, put it in `buffer`. If size of `buffer` reaches $\sqrt{Q}$, reconstruct the up-to-date version of $A[]$ by using the new updates in `buffer` and difference array – this is done in $O(N)$ – then clear `buffer`. To deal with `get(i)` command, take `sum = A[i]`, then for every pending updates which cover `i`, add its value to `sum`. This gives a $O(Q^{1.5} + Q^{0.5}N)$ solution.

Problem 34. There is a N by M grid of initially white cells, except one black cell (s_x, s_y). Then $N \cdot M - 1$ queries (x_i, y_i) are given, each representing a white cell. For each query, output the Manhattan distance to the nearest black cell and mark that cell as black. Constraints: $NM \leq 10^5$.

Solution

Naïve idea. One could either run a full BFS again after each new black cell is added, or explicitly check all existing black cells for nearest one each time we want the distance. Both methods require processing all previously updated cells repeatedly, resulting in $O((NM)^2)$ time overall.

Batching approach.

Group queries into batches of about $\sqrt{NM}$ in size. After each batch, perform a multi-source BFS from all black cells so far to get up-to-date distance table. To get the nearest black cell, look at the current distance table, then consider all the black cells pending in the buffer. The amortized complexity becomes $O(NM\sqrt{NM})$. This essentially merges the two naive approaches – a recurring theme in square-root decomposition.

2.15 Mo's Algorithm

Mo's Algorithm is a variant of *blocking*. It trivialize many range query problems, in its most basic form, it groups the queries by chunking one of their endpoint.

2.15.1 Basic Mo's – Subarray query

Problem 35. Distinct color query: Given array $A[1..N]$ of positive integers not exceeding a million. Answer Q queries, where i^{th} query is l_i, r_i, print the number of distinct elements in $A[l..r]$. $N \leq 100000$ and $Q \leq 100000$. Offline processing is allowed.

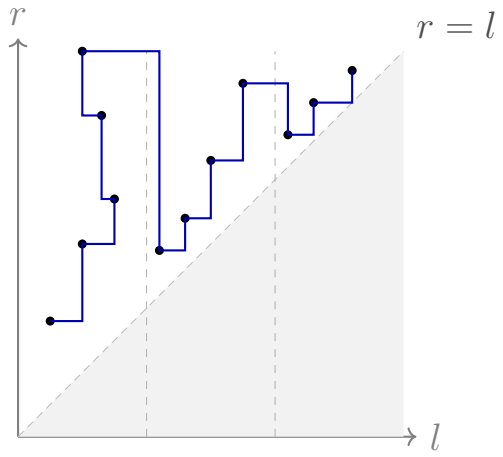


Fig. 2.10: Mo's algorithm path visualized

Solution Let us view the problem from a new perspective. Say we have data structure D supporting three operations: add one instance x, remove one instance of x, count distinct number in D. This can be implemented in $O(1)$ with an array and a counter.

Now imagine us moving on a cartesian plane where being at $(x, y); x \leq y$ means that D contains all the elements from $A[x..y]$ and nothing else. If D is empty, we can be at $(i, i-1)$ for any i. To answer all queries, we need to visit each of (l_i, r_i). We can move from (x, y) to any of the valid 4-adjacent cells in $O(1)$: to move to $(x+1, y)$, do $D.remove(A[x])$; to move to $(x-1, y)$, do $D.add(A[x-1])$; and similar for moving along OY axis. Hence, moving from a point to another takes as many moves as their Manhattan distance.

We want to travel and visit each of the queries (l_i, r_i) with sufficiently small number of moves, for the number of moves is equal to the number of operations we need to perform. The optimal path is hard to find, since that is essentially

the Hamiltonian path problem, but we can make a fairly good approximation – there exists a simple construction giving $O(Q\sqrt{N} + N\sqrt{N})$ moves.

The construction is as follows: divide the indices $(1..N)$ to blocks of size B – there are N/B such blocks. Bucket the queries by $\lfloor l_i/B \rfloor$. To process a block, move to the first point in the block – this take at most $2N$ moves. Then, move over the points in increasing order of the ordinate. For each point, we do at most B abscissa moves, and since we only move up and not down, we do at most N ordinate moves for the whole block. There are at most $N/B + 1$ blocks. Hence the total moves is upper bounded by $\frac{N}{B+1} \cdot (2N + N) + QB$. Let B be $\sqrt{N}$, then the number of moves – and time complexity of the solution – is in $O(N\sqrt{N} + Q\sqrt{N})$.

It seems convoluted, but the implementation is just a custom comparator and moving endpoints around.

```
struct Query {
    int l, r, idx;
    bool operator<(const Query& other) const {
        if (l / B != other.l / B) return l < other.l;
        return r < other.r;
    }
};

const int MAXA = 1'000'000 + 5;
int N, Q, A[MAXN], freq[MAXA], ans[MAXQ];
int distinct = 0, id_ = 0;

auto add = [&](int x) { if (++freq[A[x]] == 1) ++distinct; };
auto remove = [&](int x) { if (--freq[A[x]] == 0) --distinct; };

cin >> N >> Q;
for (int i = 1; i <= N; ++i) cin >> A[i];
vector<Query> q(Q);
for (auto &[l, r, id]: q) cin >> l >> r, id = ++id_;

B = sqrt(N);
sort(q.begin(), q.end());
```

```
23
24 int L = 1, R = 0;
25 for (auto [l, r, id] : q) {
26     while (L > l) add(--L);
27     while (R < r) add(++R);
28     while (L < l) remove(L++);
29     while (R > r) remove(R--);
30     ans[id] = distinct;
31 }
32
33 for (int i = 0; i < Q; ++i)
34     cout << ans[i] << '\n';
```

Parity-based Tiebreaking

A practical optimization of Mos algorithm is to alternate the direction in which queries are ordered inside each block of ℓ. Ordinarily, queries are sorted by block index $\lfloor \ell/B \rfloor$ and then by increasing r, which causes the right pointer to jump back to the start of the array between blocks.
If even blocks are sorted by increasing r and odd blocks by decreasing r, the traversal is likely to be more smooth. This often halves the constant factor of the algorithm on weak testsets.

2.15.2 Rollback Mo's

Sometimes, the data structure D might not support deletion – only rollback. Which means from (x, y) we can either undo the last move, or move to $(x, y+1)$ and $(x-1, y)$.
To handle such cases, we use a variant called *Rollback Mos algorithm*. The idea is to process queries grouped by their left endpoint block, just as before, but reset the whole data structure after finishing all queries of one block.
Additionally, as rightward moves are not allowed, for each block we starts at the first abscissa to the right of focused block, then temporarily extends to the left and do rollbacks on D to handle the varying left endpoints.

Problem 36 (JOI14_Historical)**.** Given array of integers $A[1..N]; 1 \leq A[i] \leq N$ and Q queries in the form (l_i, r_i). For i^{th} query find $argmax_{x \in \mathbb{Z}}\ x \cdot |\{j, A_j = x\}|$.

Solution The basic Mo's algorithm with a binary search tree can solve the problem in $O(N^{1.5} \lg N)$; however, it can be solved in $O(N^{1.5})$ with Rollback Mo's. The data structure D here is a frequency array, max value of ($x \cdot$ frequency of x), and the argmax. We cannot remove an element, for we discard the non-maximum elements. D can be reset in $O(N)$ time. The add operation, and its rollback on D take $O(1)$ time.

We solve for each block independently, resetting D afterward. To solve for the block covering interval $[start, end]$, we starts at the point $(end + 1, end)$, then process the points in that block in increasing order of ordinate – performing upward moves. We still need to handle the elements in $[l_i, end]$; we do this by performing leftward moves from $(end + 1, r_i)$ to (l_i, r_i) just before we fetch the answers. After the answer to i^{th} query is obtained, we rollback the changes until we are back at $(end + 1, r_i)$ – no rightward moves are done here.

Problem 37 (SheepDev Round 1 – C)**.** You are given a graph with N vertices and M edge (parallel edges and loops are possible). Each edge has an integer weight not exceeding 1000000000. The vertices are numbered 0 to N - 1. You must process Q queries, each in form of pair x y. You shall imagine a situation where every edges with weight less than x or more than y is removed from the graph, then print the number of connected component in the graph. Note that you are only imagining it, and no edges actually get removed.

Solution Sort the edges, then each query is limiting the graph to using only an interval of edges. Use Rollback Mo's with disjoint-set-union to solve it. Time complexity is $O(N^{1.5} \lg N)$.

2.15.3 Balancing the Time Complexity

We do $O(Q\sqrt{N})$ moves but fetch the answer merely Q times.

Problem 38. Given an array A of length N where each element is a positive integer between 1 and N. For each query (l_i, r_i), determine whether there

exists a **majority element** in the subarray $A[l_i..r_i]$, i.e. a value that occurs more than $\frac{r_i-l_i+1}{2}$ times.

Solution.

We can achieve $O(N\sqrt{N})$ total time. Assume $Q = O(N)$.

Notice that Mo's algorithm executes $O(Q\sqrt{N})$ moves but only Q heavy queries, we can afford each query on the data structure to cost $\sqrt{N}$ times more than the add and remove operations while not affecting the overall complexity.

We apply Mo's algorithm to handle the (l_i, r_i) intervals. As usual, we maintain a frequency array

$$\texttt{freq}[x] = |\{j \in [l, r] \mid A[j] = x\}.$$

Each movement of the endpoints in Mo's order adds or removes one element, so the total number of `add()` and `remove()` operations is $O(N\sqrt{N})$.

To check whether a majority element exists in the current range, we need to know whether some frequency exceeds $\frac{r-l+1}{2}$. Instead of scanning all values, we maintain a histogram over the current frequency distribution:

$$\texttt{bucket}[k] = \text{number of distinct values with frequency exactly } k.$$

When we add or remove a value x, we update its frequency and adjust two counters:

```
void add(int x) {
    int f = freq[x];
    bucket[f]--;              // one fewer element with frequency f
    blockSum[f / B]--;
    ++freq[x];
    ++bucket[freq[x]];
    ++blockSum[freq[x]];
}
```

where `blockSum[i]` stores the total number of elements whose frequencies fall in the i-th block.

All these operations are $O(1)$.

To answer a query, we simply check whether there exists any $f > \frac{r-l+1}{2}$ such that $\texttt{bucket}[f] > 0$:

```
bool hasMajority(int len) {
    int threshold = len / 2 + 1;
    int b = threshold / B;
    for (int i = b + 1; i < numBlocks; ++i)
        if (blockSum[i] > 0) reference true;
    for (int f = threshold; f < (b + 1) * B; ++f)
        if (bucket[f] > 0) return true;
    for (int f = threshold; f < min((b + 1) * B, MAXF); ++f)
        if (bucket[f] > 0) return true;
    return false;
}
```

The query operation thus costs $O(\sqrt{N})$, while updates remain $O(1)$.

Complexity Analysis.

$$O(N\sqrt{N}) \text{ moves} + Q \cdot O(\sqrt{N}) \text{ queries } = O((N+Q)\sqrt{N}).$$

The allowed $O(\sqrt{N})$ time motivates a square-root decomposition approach. We design the data structure as follows: block the integers from 1 to N into blocks of size K. On $insert(x)$ we calculate value of x as the chosen value – if that value is the maximum in its block (that is block $\lfloor \frac{x}{K} \rfloor$), we take note of that maximum in the block. On $query(a, b)$ operation, look at all blocks which lie completely in $[a, b]$, each of their maximum is an lower bound for the answer. Then we look at each index on the border individually, checking if their value is a tighter lower bound. Of course, we let $K = O(\sqrt{N})$ to obtain $O(\sqrt{N})$ complexity.

2.15.4 3D Mo's

Standard Mo's algorithm is limited to static arrays. To handle point updates, the algorithm is extended by adding a third dimension: time.

Problem 39. Dynamic distinct color query. Given array $A[1..N]$ of positive integers not exceeding a million. Answer Q command, where i^{th} command can

be either `ask(l_i, r_i)`: print the number of distinct elements in $A[l..r]$, or `upd(p_i, x_i)`: change $A[p_i]$ to x_i. $N \leq 100000$ and $Q \leq 100000$. Offline processing is allowed.

Rather than a cartesian plane, now we imagine a 3D space. The corresponding state of point (x, y, z) is that D contains only the elements from $A[x..y]$ after the z^{th} command. Moving parallel to OX and OY axes is done the same way as in static distinct color query. To move in the direction of OZ axis, we can add and remove some elements from D. There are $O(1)$ such operations we have to do to move one step.
Since we can still move efficiently, it is a matter of finding a short enough path. One way to do that is to divide both the OX and OY axes to blocks of size B. We then process each "tower" – $B \times B$ square which expands infinitely in OZ axis separately; at the start of each tower we reset D – move it to the point with lowest applicate, then move our state as needed, visiting all points contained in that tower in increasing order of applicate.
The time complexity is $O((N/B)^2 \cdot Q + Q \cdot B)$. That is minimized when $B \approx N^{2/3}$: $O(N^{5/3} + QN^{2/3})$. If Q is $O(N)$ then it is nicely $O(N^{5/3})$. Guess what the time complexity of 4D Mo's would be!

Problem 40. `SPOJ - XXXXXXXX`

2.15.5 4D Mo's

To deal with a 14-dimensional space, visualize a 3-D space and say 'fourteen' to yourself very loudly. Everyone does it.

Geoffrey Hinton

Problem 41 (Codeforces 1767F)**.** Given a tree of order N rooted at vertex 1. Each vertex has integer v_i written on it. Answer Q queries; the i^{th} query is (u_i, v_i). To answer it, you shall collect all the vertices in subtrees u_i and v_i then

list the number written on those vertices – a vertex is listed twice if it is in both subtree u_i and v_i. Then print the lowest mode in the list.
$1 \leq N, Q, v_i \leq 200000$.p

Solution Run a depth-first-search on the tree to obtain its DFS order. Consider a 4D space (do not try imagining it). A point $P = (x_i, y_i, z_i, w_i)$ correspond to a state where the integer written on each vertex with discovery time in $[x_i, y_i]$ is put in the *list*, and then the integer written on each vertex with discovery time in $[z_i, w_i]$ is again put in the *list*. Apply Mo's algorithm. The data structure, which need to support adding and removing one instance of an integer v_i, and answering the minimum mode, can be implemented in amortized constant time; the detail for its implementation will be discussed later. As we have the data structure, now we implement Mo's algorithm on four dimensional space. We block the first three axes. There will be in total $(N/B)^3$ tesseract-shaped buckets to classify our points. The cross section of the each of those tesseract against OXYZ space is a cuboid with side length B. and sort points in each by their fourth coordinate. To move to a new 1tesseract we can either reset the data structure or move back along the fourth dimension – both take $O(N)$. The time complexity adds up to $O(NB + N^4/B^3)$. Selecting $B = N^{.75}$ minimizes it.

Exercise 42. How to design the data structure? Hint: `add(x)` takes constant time, `remove(x)` takes amortized constant time, and `getmode()` can take up to $O(N^{.75})$.

2.16 Dynacon – Segment tree on Timeline

Offline removal in non-amortized insert-only structures with an extra log factor

The name *Dynacon* is not unanimous – many calls it that because applying the trick to a disjoint-set union solves the dynamic connectivity problem.

Say we have data structure `D` and need to support Q operations of these types:

`insert(y)`, `remove(i)`, `ask(x)` – where `y` and `x` are arbitrary data, and `remove(i)` undo the insertion done in i^{th} query.
Let us build a segment tree with Q leaves, each vertex of the tree stores a list (as in dynamic array) of data associated with insertion queries. For each insertion query, find its lifetime – when it gets removed. If it never gets removed, consider it removed at time $Q + 1$. For insertion query with data y to be inserted and lifetime $[l, r]$, we append y to each of the $O(\lg Q)$ segment tree vertices making up the lifetime interval. Once we do that for all insertion query, we can find the answer for all `ask` queries. Do a *depth-first-search* down the segment tree, starting from the root. When entering a vertex, apply all the updates attached to that vertex. Before leaving a vertex, *undo (stack-wise)* all the updates attached to that vertex. All non-amortized data structure can support the undo operation nicely – just revert every modification made to the computer's memory. After entering a leave vertex associated with timestamp *t*, if t^{th} query is an `ask` query, do a query on the current data structure and output the answer (or memorize it). Of course, any insertion query is done $O(\lg Q)$ times, so this adds a log factor to the time-complexity.

Problem 43 (SPOJ – DYNACON2)**.** Dynamic Connectivity Problem: maintain a graph; support adding edge, removing edge, and asking whether a u-v path exists.

Solution The relation to Dynacon is obvious: each edge lives in a certain time interval. We can use union-by-rank disjoint-set-union, which supports undo, to maintain the connectivity.

```
1  int answer[Q];
2  vector<pair<int, int> > t[Q * 4];
3  void add(int v, int l, int r, int x, int y, pair<int, int> edge) {
4    if (r < x || y < l) return;
5    if (x <= l && r <= y) {
6      t[v].push_back(edge);
7      return;
8    }
9    add(v * 2 + 1, l, (l + r) / 2, x, y, edge);
10   add(v * 2 + 2, (l + r) / 2 + 1, r, x, y, edge);
```

```
11  }
12  UndoableDSU ds;
13  void dfs(int v, int l, int r) {
14    for (auto [a, b]: t[v]) ds.addedge(a, b);
15
16    if (l == r) {
17      if (query_type[l] == "conn")
18        answer[l] = ds.is_connected(ask_a[l], ask_b[l]);
19    } else {
20      dfs(v * 2 + 1, l, (l + r) / 2);
21      dfs(v * 2 + 2, (l + r) / 2 + 1, r);
22    }
23
24    for (auto _ : t[v]) ds.undo();
25  }
```

Problem 44. Minimum range spanning tree: given a weighted graph, find the minimum k such that there exists a spanning tree where the difference between maximum and minimum weight does not exceed k.

Solution There is an $O(N \lg N)$ solution with link-cut tree, but $O(N \lg^2 N \lg A)$ can be done with Dynacon when A is range of edge weights. First, consider this easier version: given k, is there Y such that the graph is connected only by edges with weight in $[Y, Y + k]$? Notice that we can consider only the value of edge weights as Y.

If we sort the edges, for each i, you find out what is maximal $j \geq i$ such that we can use all edges from i to j if i^{th} edge is the minimum-weighted one; let that j be $f(i)$. Now we have N intervals of edges where both the starts and ends are increasing – because f is increasing. We shall check if any of those intervals are valid.

To do that, consider each interval as a timestamp. Each edge will be alive in a consecutive time interval. Use Dynacon to find out if the graph is connected in any timestamp. If it is, the current value of k is an upper bound of the answer of our original problem. Do binary-search on answer!

Problem 45 (Codeforces 1217F)**.** Maintain an initially empty graph on N

vertices under M queries. The queries are *forced online* using a variable last $\in \{0, 1\}$, which stores the result of the most recent connectivity check. For a query on input coordinates x and y, the actual vertices are mapped using the formula $u = (x + \text{last} - 1) \bmod N + 1$ and $v = (y + \text{last} - 1) \bmod N + 1$. The operations are:

1. **Toggle Edge:** Add the edge (u, v) if it does not exist, or remove it if it does.
2. **Connectivity Check:** Determine if u and v are connected. Update last with the result.

The objective is to output the results of all connectivity checks.

Solution This problem, despite its "online" encoding, can be solved by converting it to an offline Dynamic Connectivity Problem. The constraint that last can only be 0 or 1 means that every query affects one of only two possible edges. This *weakly encoded* nature allows us to preprocess all potential edge events. The hard part is to handle the edge ambiguity.
We first identify all unique potential edges. For each of the M queries, we consider both possibilities for the last variable (last $= 0$ and last $= 1$), generating up to $2M$ total potential events. We then group all events for the same unique edge $e = (u, v)$. If the edge e has potential to be toggled at queries $x_1, x_2, \ldots, x_k$, we compute its lifetime intervals: $[x_1, x_2), [x_2, x_3), \ldots, [x_{k-1}, x_k), [x_k, M + 1]$. These intervals are then mapped to the nodes of the segment tree.
The key modification is the use of a global boolean array, active$[\cdot]$, which tracks the true state (present or absent) of every unique potential edge. The DFS traversal of the segment tree proceeds as follows:

1. **Internal Node (v):** When entering a node v, we iterate over all potential edges whose lifetime intervals cover v's time span. We only apply the edge update to the DSU (via ds_unite) if the edge is currently marked as *active* in the global active$[\cdot]$ array. This ensures only truly existing edges are part of the structure at this time.

2. **Leaf Node (t):** When reaching a leaf node corresponding to query t:
   - If t is a *connectivity check* (Type 2), we perform the DSU query and update the global last variable for all subsequent queries.
   - If t is an *edge toggle* (Type 1), we use the current, correct value of last to determine the real edge e^*. We then toggle the state of e^* in the global active$[\cdot]$ array. This is the moment where the online query is correctly decoded and its effect is registered for the rest of the offline process.
3. **Rollback:** Just as normal Dynacon, when exiting node v we use the DSU rollback mechanism to revert all temporary insertions, maintaining the correct state for the sibling branch.

The overall time complexity is $O(M \lg N \lg M)$. Refer to **??** for the code.

2.17 Parallel Binary Search

Problem 46 (CSES New Road Queries)**.** Given graph of order N and size M where each edge is labeled from 1 to M. Answer Q queries (u_i, v_i). For i^{th} query print the minimal x such that there exists a path from u_i to v_i using only edges with label not exceeding x.

Solution

A funny solution is to perform binary search for each query, and check the connectivity with a disjoint-set-union. This takes $O(QM \lg M\alpha(N))$ – each query performs $O(\lg M)$ layer of checking, in each layer up to M edges is added to the DSU.

Now we can reorder the process a bit; Check connectivity first layer of every queries – which will share same `mid = M / 2`. Then check connectivity of second layer of every queries – some to be checked at `M / 4` and some at `3M / 4`. Notice that if we add edges in the order of their labels, we only need to go over the edge list once and not twice. i.e. we add edges with label not exceeding $M/4$ then check all the queries that shall be checked at that point, and then

add remaining edges with label not exceeding $3M/4$ then check all the queries that shall be checked at that point. This way, we spend $O((Q+M)\alpha(N))$ on each layer of the binary search. Continue the process until we know the answers to all queries. The total complexity is $O((Q+M)\alpha(N)\lg M)$.

In Figure 2.13, we have four queries A, B, C, D. Each horizontal dashed line represents one *layer* of the parallel binary search – that is, one complete *left-to-right sweep* of the edge list by the disjoint-set-union (DSU), adding edges in increasing order of their labels.

At the first layer, all queries check whether using the first two edges is sufficient to connect their respective vertex pairs. After one DSU sweep, we find that this range suffices for queries A and C, but not for B and D.

In the second layer, the remaining searches refine their ranges: queries A and C now test whether only the first edge is enough, while B and D test whether the first three edges suffice. We again perform a single DSU sweep – adding the first edge, checking A and C, then adding the second and third edges, and checking B and D.

Exercise 47 (POI11 Meteors)**.**

2.18 CDQ

More dimensions!!!

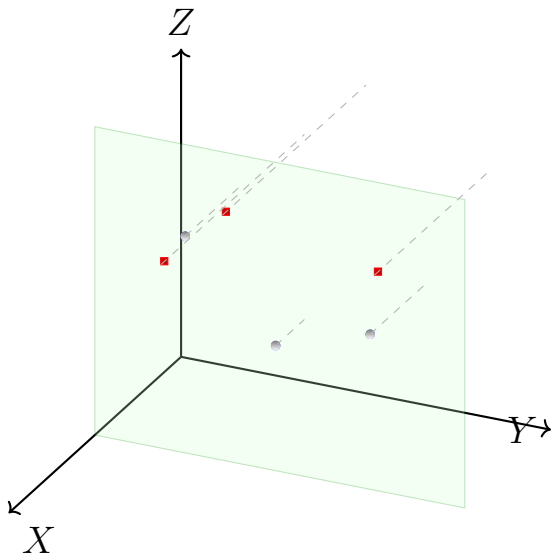
Problem 48. There exist $N \leq 10^5$ points $P_i = (x_i, y_i, z_i) \in {\mathbb{Z}_+}^3$. We define relation $<$ as follows: $(x,y,z) < (x',y',z') \iff x < x' \wedge y < y' \wedge z < z'$. For each $1 \leq i \leq N$, let $f(i)$ be number of points $1 \leq j \leq N$ such that $P_j < P_i$. Find $f(i)$ for all $1 \leq i \leq N$. To save us from the annoying details, assume that all the abscissas are pairwise distinct. This is the 3D partial order problem.

Solution 1: Bashing 2D point add rectangle sum data structure. Iterate over the points in increasing order of x_i. To find $f(i)$, query the sum of rectangle $((0,0),(y_i,z_i))$. Then add one to point (y_i,z_i). Time and space complexity: $O(N\lg^2 N)$. This is fine but has huge constant and memory usage.

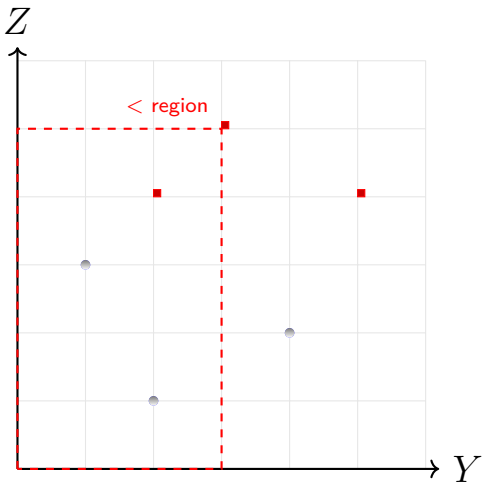
Solution 2: CDQ. Start by sorting and points in increasing order of abscissa. Let's assume P is already sorted as such. We know that $i < j$ is a necessary condition for $P_i < P_j$. This motivates a divide-and-conquer approach. To find the answer for points in P, we divide P into left and right part of balanced size. Now for each point in the right part we consider how many points in the left part is less than itself. Once we do that, the left and right part becomes completely isolated and we can solve the two subproblems.

The usual divide-and-conquer process is to *combine* the results of the two subproblems; however, here it is more like *count contribution between two subproblems and divide*. We do not relate the results of two subproblem in any way. Now, to solve the *count contribution* part – what we shall solve is essentially: given set A and B of points (y_i, z_i) in a cartesian plane. For all points in B (the right part), count the number of points in A which is less than it. This is a standard problem which can be solved by offline processing and one-dimensional Fenwick tree. Time complexity remains $O(N \lg^2 N)$. However, space complexity is merely $O(N)$, for we use one dimensional range-add-range-sum structure here.

Generally, CDQ is highly useful in cutting off one dimension in a geometric process at a cost of a log factor.



1. $\mathbb{Z}_+^3$: Divide by X



2. $\mathbb{Z}_+^2$: 2D Subproblem

Fig. 2.14: CDQ visualization. Top: points in $\mathbb{Z}_+^3$ split by $X = c$ (green plane) into two subsets. Bottom: projection onto OYZ-plane forming the 2D dominance subproblem.

Exercise 49 (APIO19_street)**.**

Exercise 50. Invent an $O(N \lg^3 N)$ solution to 4D partial order problem.

Exercise 51 (DarkBzoj3295)**.**

Exercise 52 (ZJc571)**.**

2.19 Minimum Stack – Minimum Deque

push, pop, getmin

Let us design a stack which support these operations: `push(x)` `pop()` `getmin()`. Where `getmin()` returns the minimum element in stack. This can be implemented in same complexity as standard stack. Maintain two stacks A, B; on a `push(x)` operation, push $min(x, B.top)$ – or x if B is empty – to B and push x to A. On `pop()` operation, do a pop on both A and B. On `getmin()` operation, return $B.top$. Let us call this data structure min-stack.
There is a standard technique for implementing a queue with two stacks. Maintain stacks A and B. to do an enqueue operation, push to A. To dequeue, if B is empty, repeated push the top of A to B then pop A, until A is empty. Then return top of B. The amortized time complexity is constant time per operation, for each element can only get moved from A to B once. The intuition is that A stores some elements from the rear end of the queue, and B stores the remaining elements.
If we use two min-stacks to implement a queue, we achieve a min-queue. Is it possible to get min-deque? Yes, it turns out we can implement an efficient deque with two stacks. It is almost identical to building a queue – except we only move half the elements to the empty stack instead of the entirety of the other stack. If we want to pop from B and it is empty, move half top of A to it. If we want to pop from A and it is empty, move half top of B to it. Now, by using min-stacks to build a deque, we achieve efficient amortized min-deque!

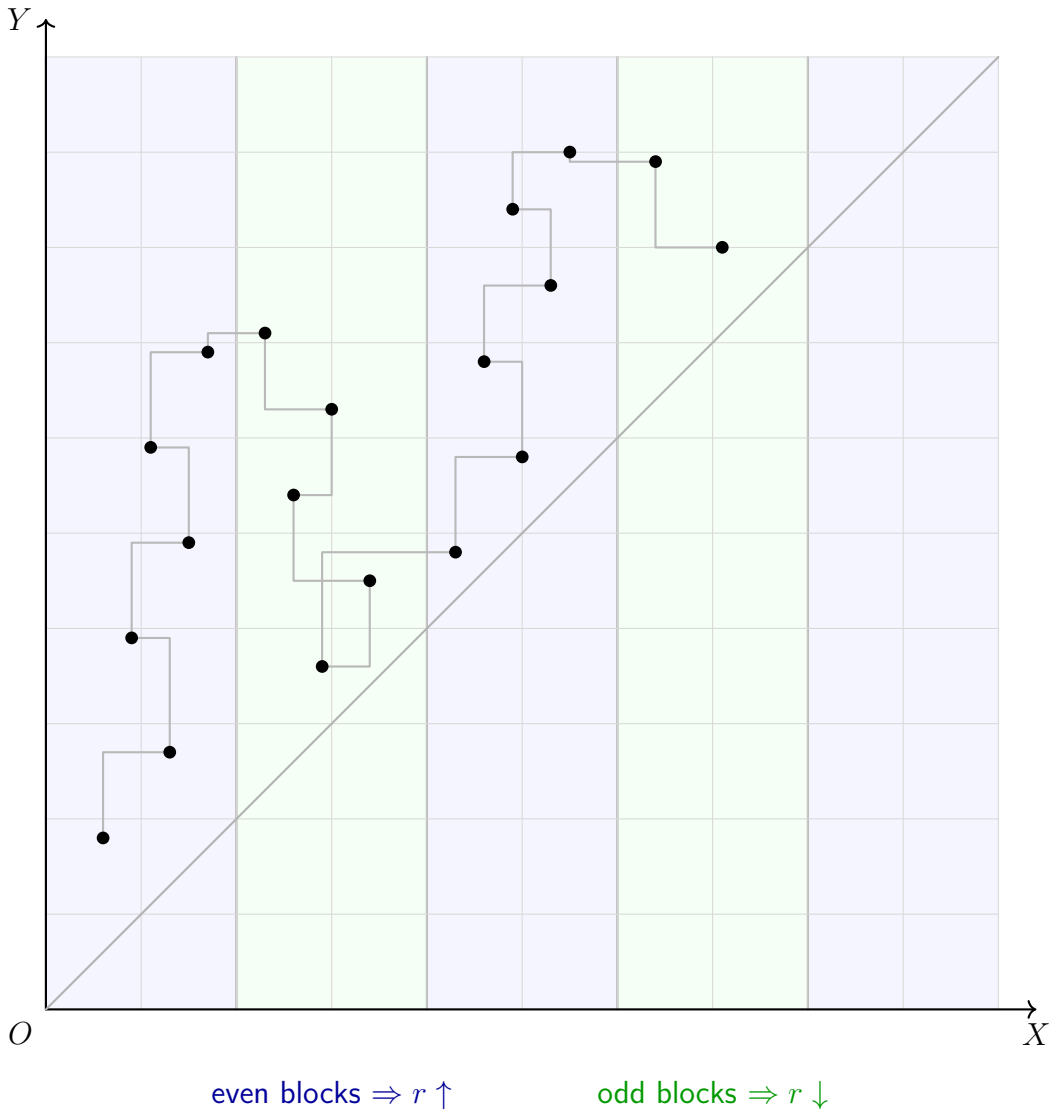


Fig. 2.11: Parity-based tiebreaking

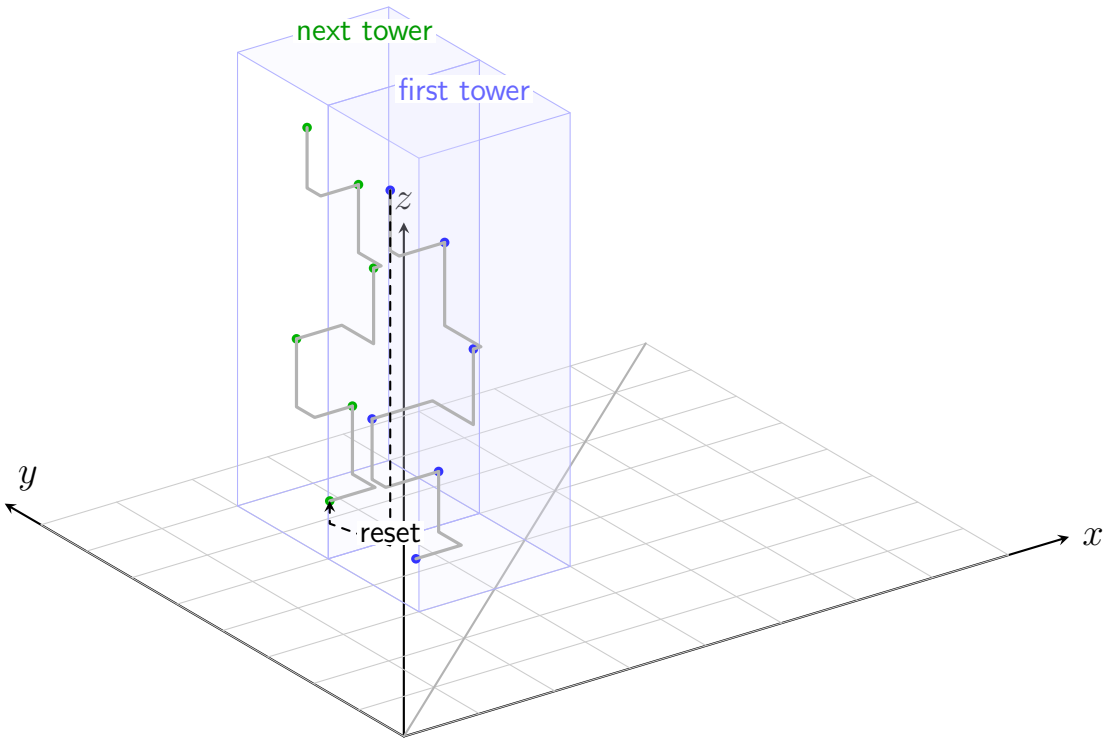


Fig. 2.12: 3D Mos algorithm in OXYZ space. Each query (ℓ, r, t) is represented as a point. The OXY plane is partitioned into blocks of size $B \times B$ which extends in OZ axis, forming vertical *towers*. Within each tower, enlarged query points are processed along a light-gray path.

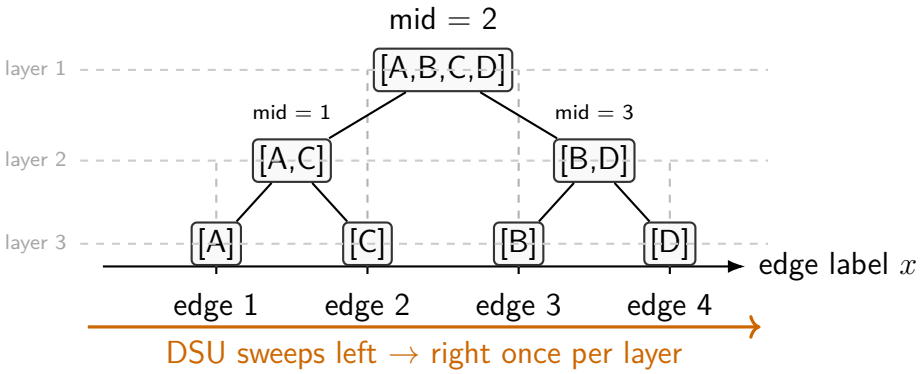


Fig. 2.13: Parallel Binary Search with four edges and queries.

Bibliography

[1] AtCoder Inc. Atcoder programming contest platform. https://atcoder.jp/, 2025. Accessed 2025-01-31.

[2] Codeforces Community. Codeforces competitive programming platform. https://codeforces.com/, 2025. Accessed 2025-01-31.

[3] CP-Algorithms Contributors. Cp-algorithms: Algorithms for competitive programming. https://cp-algorithms.com/, 2025. Translated and maintained by competitive programming community; Accessed 2025-01-31.

[4] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1), 1989.

[5] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3), 1994.

[6] International Olympiad in Informatics Committee. Official ioi website. https://ioinformatics.org/, 2025. Accessed 2025-01-31.

[7] Luogu Community. Luogu — chinese olympiad in informatics online judge. https://www.luogu.com.cn/, 2025. Accessed 2025-01-31.

[8] OI Wiki Contributors. Oi wiki — knowledge base for olympiad in informatics. https://oi-wiki.org/, 2025. Accessed 2025-01-31.

[9] OJ.UZ Team. Oj.uz — online judge for olympiad problems. https://oj.uz/, 2025. Accessed 2025-01-31.

[10] USACO Guide Team. Usaco guide — structured learning for competitive programming. https://usaco.guide/, 2025. Accessed 2025-01-31.