SEGMENT TREES

&

INTERVAL TREES

Aditya Chandran 201201115

Arvind Menon 201230107

SEGMENT TREES

A segment tree is a data structure for storing a set of intervals :

$I = \{[x_1 , x_1], [x_2 , x_2], . . . , [x_n , x_n]\}$

and can be used for solving problems e.g. concerning line segments.

Let $p_1 , . . . , p_m$, $m \leq 2n$, be the ordered list of distinct endpoints of

the intervals in I. The ordered sequence of endpoints $p_1 , . . . , p_m$ par–

titions the real line into a set of atomic intervals

$(-\infty, p_1), [p_1 , p_1], (p_1 , p_2), [p_2 , p_2], . . . , (p_{n-1} , p_n), [p_n , p_n], (p_n , \infty)$

Let int(v) denote the interval corresponding to node v. With each

node v we store a set $I(v) \subseteq I$: Interval $[x, x]$ is stored in I(v) if and

only if –

1. $int(v) \subseteq [x, x]$ and,
2. $int(parent(v)) \subseteq [x, x]$

A segment tree on n intervals uses O(n log n) storage.

ALGORITHM INSERTION :

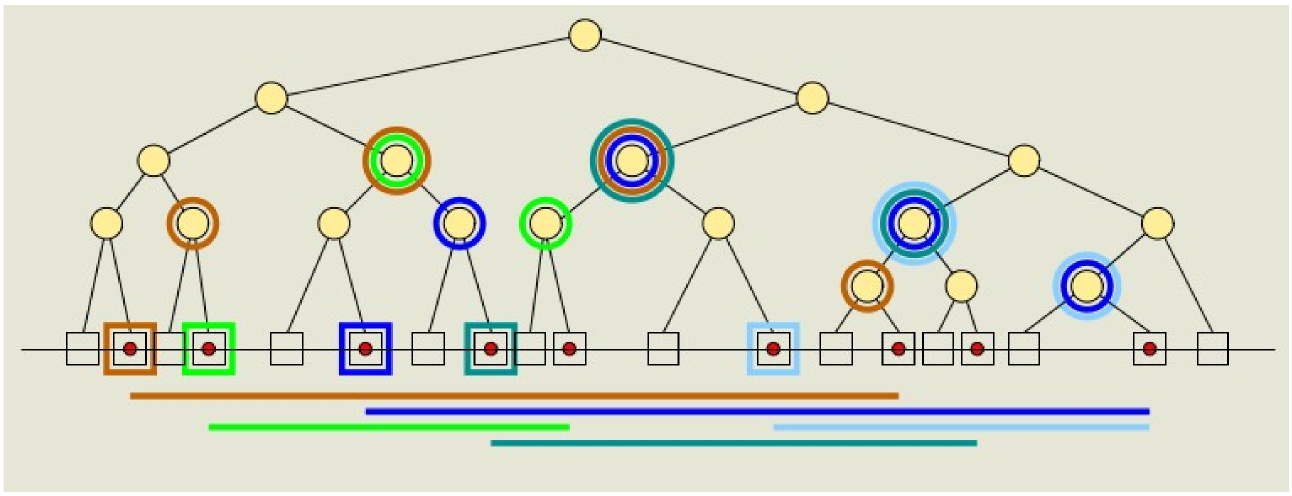
```
INSERT SEGMENT TREE(v,[x, x])
    if int(v) ⊆ [x, x ]
        then add [x, x ] to I(v)
    else if int(lc(v)) ∩ [x, x ] = 0
        then INSERT SEGMENT TREE(lc(v), [x, x ])
        if int(rc(v)) ∩ [x, x ] = 0
        then INSERT SEGMENT TREE(rc(v), [x, x ])
```

A segment tree for a set of n intervals can be constructed in O(n log n) time.

ALGORITHM QUERY :

```
QUERY SEGMENT TREE(v, qx)
    Report all the intervals in I(v)
    if v is not a leaf
        then if qx ∈ int(lc(v))
            then QUERY SEGMENT TREE(lc(v), qx )
            else QUERY SEGMENT TREE(rc(v), qx )
```

Using a segment tree, we can report all k intervals that contain a query point qx in time $O(k + \log n)$.



INTERVAL TREES

An interval tree is an ordered data structure to hold intervals.

We start by taking the entire range of all the intervals and dividing it in half at *x_ center* (in practice, *x_ center* should be picked to keep the tree relatively balanced). This gives three sets of intervals, those completely to the left of *x_ center* which we'll call *S_ left*, those completely to the right of *x_ center* which we'll call *S_ right*, and those overlapping *x_ center* which we'll call *S_ center*.

The intervals in *S_ left* and *S_ right* are recursively divided in the same manner until there are no intervals left.

The intervals in S_center that overlap the center point are stored in a separate data structure linked to the node in the interval tree. This data structure consists of two lists, one containing all the intervals sorted by their beginning points, and another containing all the intervals sorted by their ending points.

Construction requires O(n log n) time, and storage requires O(n) space.

The result is a tertiary tree with each node storing:

- A center point
- A pointer to another node containing all intervals completely to the left of the center point
- A pointer to another node containing all intervals completely to the right of the center point
- All intervals overlapping the center point sorted by their beginning point
- All intervals overlapping the center point sorted by their ending point

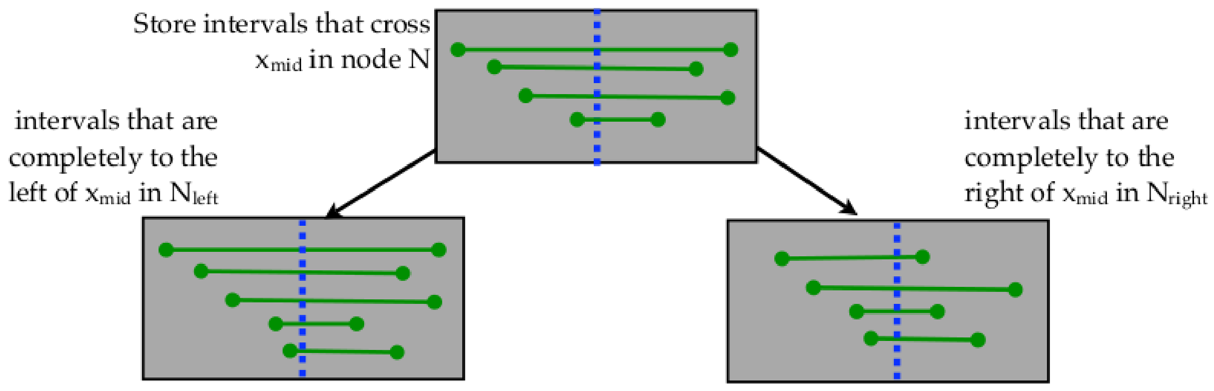
Searching intervals at current node

• Store each interval in two sorted lists stored at node:

  – List L sorted by increasing left endpoint

  - List R sorted by decreasing right endpoint

- Search list depending on which side of xmed the query is on:
  - If xq < xmed then search L, output all until you find a left endpoint > xq.
  - If xq ≥ xmed then search R, output all until you find a right endpoint < xq.
  - For each tree node, *x* is compared to *x_ center*, the midpoint used in node construction above.
  - If *x* is less than *x_ center*, the leftmost set of intervals, *S_ left*, is considered. If *x* is greater than *x_ center*, the rightmost set of intervals, *S_ right*, is considered.
  - This process continues until a leaf node is reached.


Store intervals that cross x_{mid} in node N
intervals that are completely to the left of x_{mid} in N_{left}
intervals that are completely to the right of x_{mid} in N_{right}

THEORETICAL RUN-TIME ANALYSIS

Segment Trees vs. Interval Trees :

	Segment trees	**Interval trees**
Storage	$O(n \log n)$	$O(n)$
Construction	$O(n \log n)$	$O(n \log n)$
Querying	$O(\log n + k)$	$O(\log n + k)$

DATA SETS

Assumption : The number of queries is much larger than the number of segments.

INPUT FORMAT :

The first line contains and integer N which denotes the number of segments.Next N lines contain two integers which are the end points of each segment.

This is followed by an integer M which denotes the number of queries. Next M lines contain a single integer which is the poit to be queried.

OUTPUT FORMAT :

The output consists of two lines for each query.

query :

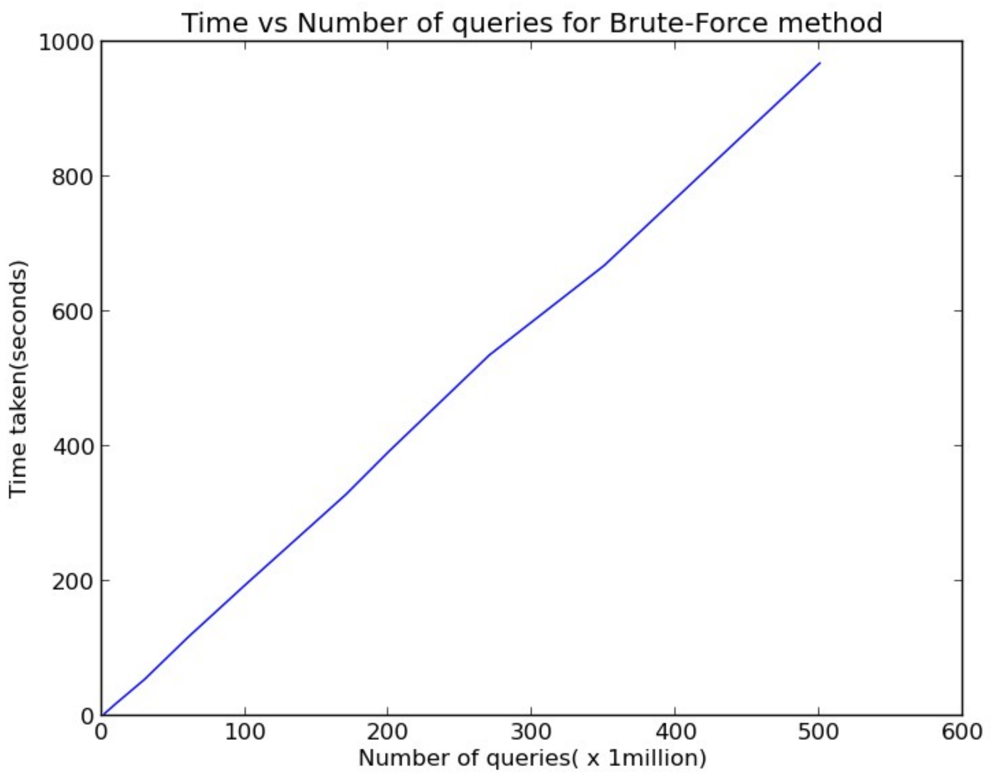
Ans :

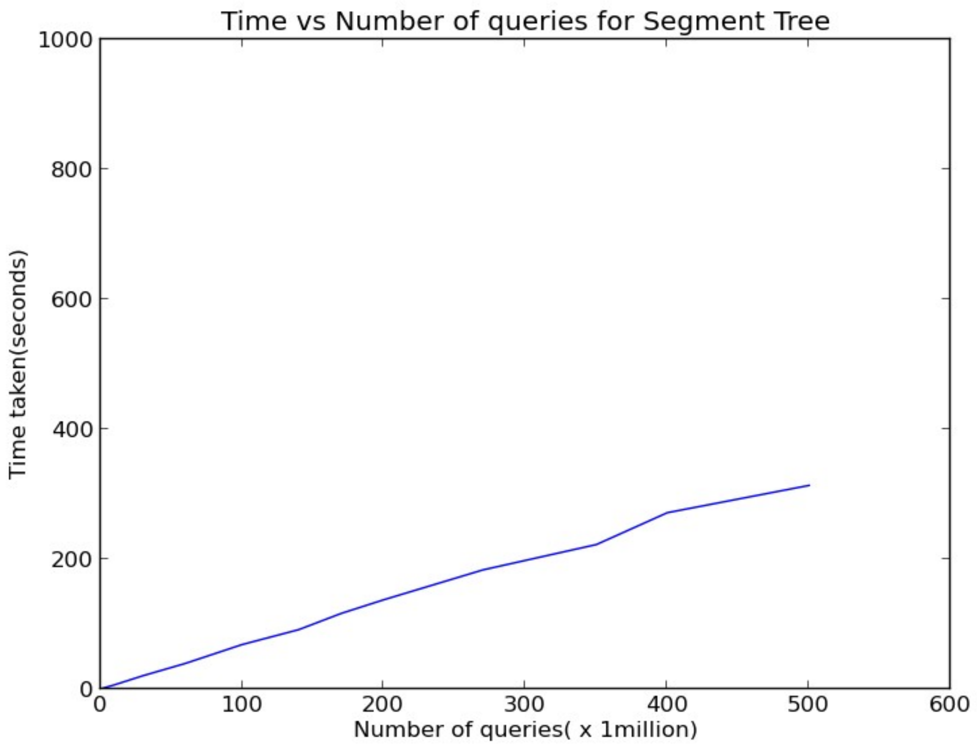
Ans is the number of segments which contain the queried point.

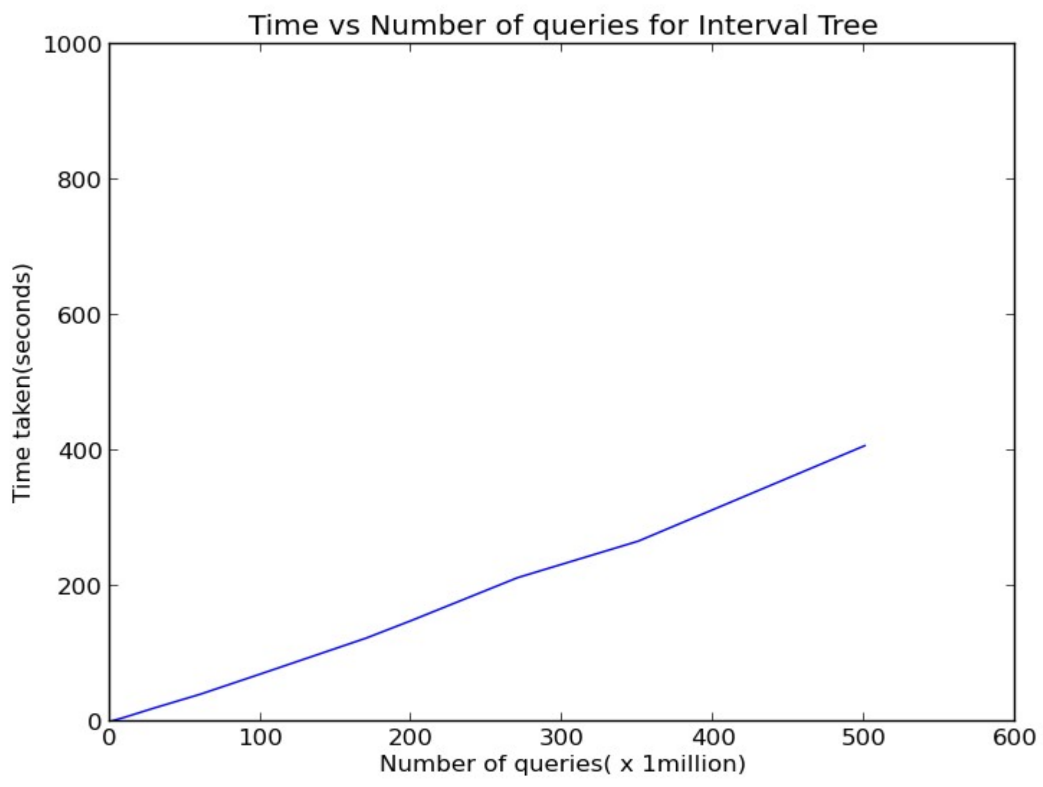
CONSTRAINTS :

$N < 1000$

GRAPHS OF NUMBER OF QUERIES vs TIME







Note :

- The number of segments was kept constant and the the number of queries was varied from 2 million to 500 million.
- The same test cases were used for interval trees, segment trees and brute force.
- The graphs were plotted using python.
- The programs output the results for the queries and stores the runtime for the queries in a separate file.
- The same test case generator (test_generator.c) was used to create multiple datasets by changing the number of queries each time (by changing variable : 'y').