Solution

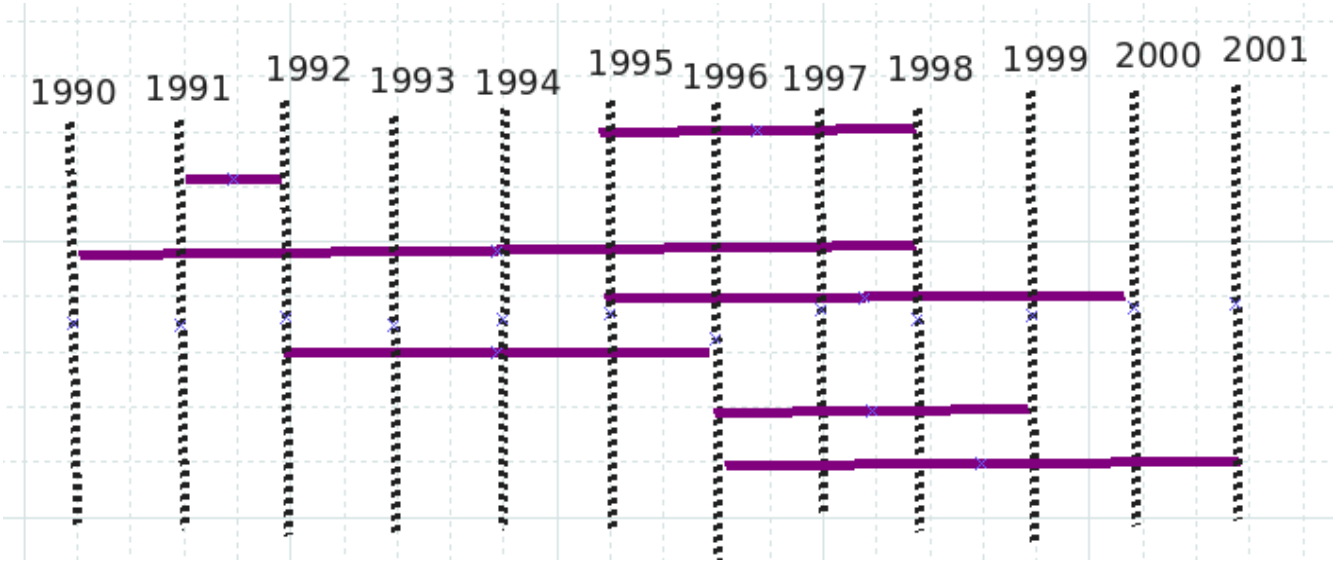
1854: Maximum Population Year on leetcode

problem link: https://leetcode.com/problems/maximum-population-year/

For time complexity, we gonna take $N = logs.size()$, $M = death\ year - birth\ year$

I gonna take this input sample:

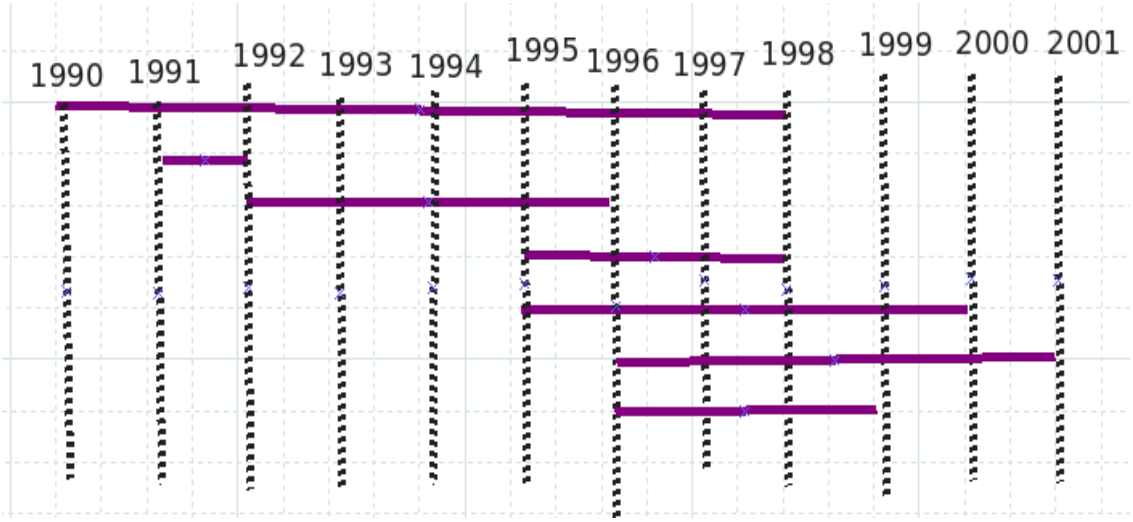
logs = {{1995, 1998}, {1991, 1992} , {1990, 1998}, {1995, 2000}, {1992, 1996}, {1996, 1999}, {1996, 2001}}.



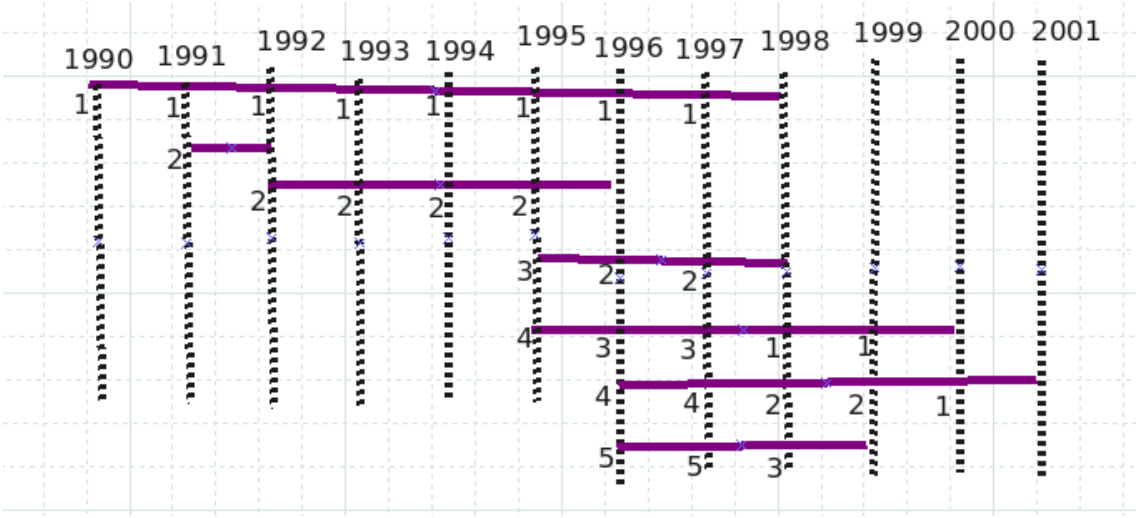
There are different approaches, but there is a naive approach and an optimal approach. It’s pretty much to know both of them because it can help you to resolve many other problems of this kind.

Naive algorithm: O(N * M) = O(N²)

Sort all segments by left value(first) value, here the birth year.



then, for each pair $birth_i, death_i$ ($1 \leq i \leq \text{logs.size()}$), add 1 for all years $year_i \in [birth_i, death_i - 1]$



We obtain this map:

{{1990: 1}, {1991: 2}, {1992: 2}, {1993: 2}, {1994: 2}, {1995: 4}, {1996: 5}, {1997: 5}, {1998: 3}, {1999: 2}, {2000: 1}}

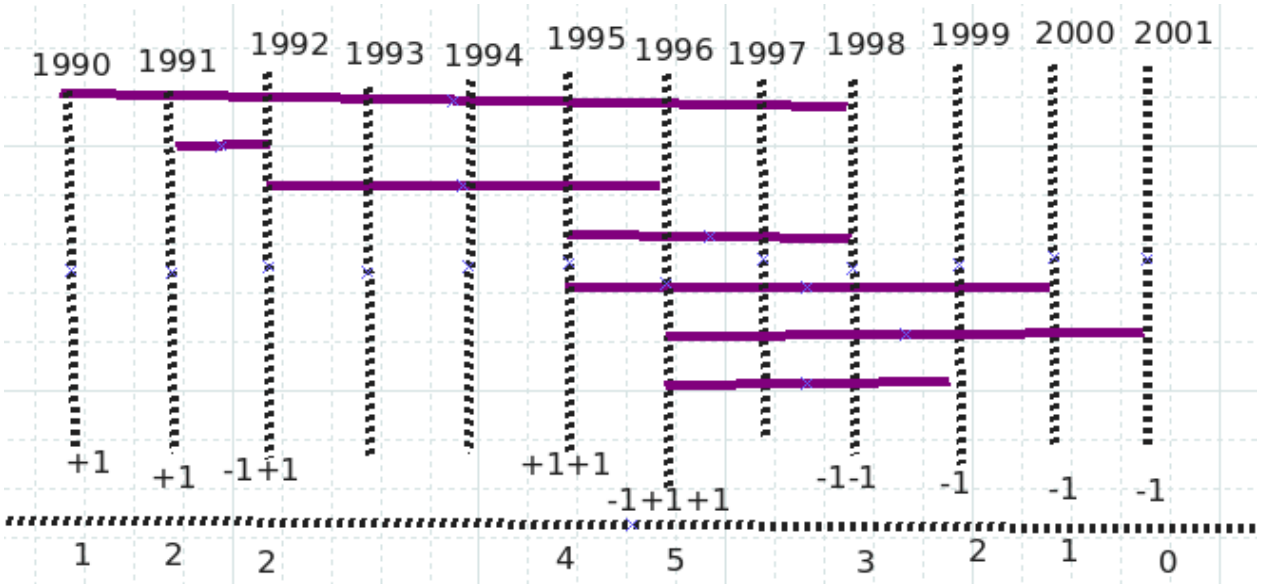
The **earliest** year with the **maximum population** is 1996 with 5 .

C++ code: O(N * M)

```
int maximumPopulation(vector<vector<int>>& logs) {

        map <int, int> count;

        for (int year = 1950; year <= 2050 ; ++year)
            count[year] = 0;

        //O(N * M)
        for (auto v: logs) {
            for (int year = v[0] ; year < v[1] ; ++year) {
                // years are inserted in ascending order in the map.
                count[year]++;
            }
        }

        int max_year = 0;
        int cnt_max = 0;
        for (auto cnt: count) {
            if (cnt_max < cnt.second) {
                    cnt_max = cnt.second;
                    max_year = cnt.first;
                }
        }

        return max_year;
    }
```

Optimal algorithm: O(N log N)

The idea is instead of counting all the population in the whole range [birth_i, death_i[($birth_{year} \leq i < death_{year}$), and if we get a close look at the first approach, we can just count and update the population just at a the *birth year* and the *death year*, by adding $+1$ to the *birth year* and -1 to the *death year*:



we get the same map as the naive approach :) :

{{1990: 1}, {1991: 2}, {1992: 2}, {1993: 2}, {1994: 2}, {1995: 4}, {1996: 5}, {1997: 5}, {1998: 3}, {1999: 2}, {2000: 1}, {2001: -1}}

C++ code: O(N log N)

```
int maximumPopulation(vector<vector<int>>& logs) {

        map<int, int> dp;
        for (auto v : logs){
            dp[v[0]]++;
            dp[v[1]]--;
        }

        int cnt = 0;
        int max_year = 0;
        int cnt_max = 0;
        for (auto p : dp){
            cnt += p.second;
            if (cnt_max < cnt){
                cnt_max = cnt;
                max_year = p.first;
            }
        }

        return max_year;
    }
```

According to the C++ reference the map insertion is logarithmic:
https://www.cplusplus.com/reference/map/map/insert/

Complexity

If a single element is inserted, logarithmic in size in general, but amortized constant if a hint is given and the *position* given is the optimal.

C++98 C++11

If N elements are inserted, `Nlog(size+N)`.
Implementations may optimize if the range is already sorted.

so:

O(N log 2N) = O(N log N + N log 2) = O(2 N log N) = O(N log N)

```
map<int, int> dp;
for (auto v : logs){
    dp[v[0]]++;
    dp[v[1]]--;
}
```

O(2N) = O(N)

```
for (auto p : dp){
     cnt += p.second;
      if (cnt_max < cnt){
          cnt_max = cnt;
          max_year = p.first;
      }
}
```

If you have a doubt that a map in C++ is linear, you can change the data structure an array which is sorted in $O(N \log N)$

C++ code with vector: O(N log N)

```
int maximumPopulation(vector<vector<int>>& logs) {
        vector<pair<int, int>> dp;
    //O(N)
    for (auto v: logs) {
        dp.push_back({v[0], 1});
        dp.push_back({v[1], -1});
    }

    // O(N log N)
    sort(dp.begin(), dp.end());


    int cnt = 0;
    int max_year = 0;
    int cnt_max = 0;

    //O(N)
    for(auto p: dp) {
        cnt += p.second;
        if (cnt_max < cnt) {
            cnt_max = cnt;
            max_year = p.first;
        }
    }

    return max_year;
```

http://www.cplusplus.com/reference/algorithm/sort/

Complexity

On average, linearithmic in the distance between *first* and *last*: Performs approximately $N*\log_2(N)$ (where *N* is this distance) comparisons of elements, and up to that many element swaps (or moves).