B: Can of Worms

There is an old adage about opening a can of worms. A lesser known adage is one about shooting a can of exploding worms with a BB gun.

Imagine we place some cans of exploding worms on a long, straight fence. When a can is shot, all of the worms inside will explode. Different types of worms have different blast radii. Each can contains only one kind of worm.

When a can explodes, if another can is in the blast radius, then that can will also explode, possibly creating a chain reaction. Each can explodes only once. This process continues until all explosions stop.

For each can, suppose that it is the only can shot. How many cans in total will explode?

Input

There will be several test cases in the input. Each test case will begin with a line with a single integer n ($1 \le n \le 100{,}000$) representing the number of cans on that fence. Each of the next n lines will have two integers x ($-10^9 \le x \le 10^9$) and r ($1 \le r \le 10^9$), where x is the location of the can on the fence and r is the blast radius. No two cans will occupy the same location. The input will end with a line with a single 0.

Output

For each fence, print n integers on a single line separated by single spaces. The i^{th} integer represents the number of cans that will explode if the i^{th} can is the one that is shot. Output no extra spaces, and do not print any blank lines between outputs.

Sample Input	Sample Output

```
3
4 3
-10 9
-2 3
12
2 2
7 7
10 1
19 3
23 12
29 8
33 1
35 17
39 2
40 1
46 11
52 3
0
```
	```
	1 2 1
	1 3 1 1 9 9 1 9 2 2 9 1
	```

```
(*
  Solution to the "can of worms" problem from the 2013 University
  of Chicago invitational programming contest.
  http://icpc.cs.uchicago.edu/invitational2013/

                       Danny Sleator, April 2013

  Okay, here's an observation about B.

  First think of the cans as indices, and now each can
  has a range of can indices that it triggers.  So make
  arrays lo[] and hi[] to represent these ranges.

  Define an "expand" operator that (for all i) simply replaces lo[i]
  by the lowest low in the range [lo[i], hi[i]].  (Similar for hi[i]).
  You can do this efficiently by using a range-min data structure
  (and range-max data structure).

  If we iterate the expand operator until there are no
  changes, we're done.

  The kth iterate of expand is inferring all chains of up to
  2^k  in length.  Therefore it must terminate in O(log n)
  iterations.

  I implemented this using Namit's range min trick.  It solves
  the test data in 10 seconds.
*)

let read_int () = Scanf.bscanf Scanf.Scanning.stdib " %d " (fun x -> x)
let read_long () = Scanf.bscanf Scanf.Scanning.stdib " %Ld " (fun x -> x)

let ( ++ ) a b = Int64.add a b
let ( -- ) a b = Int64.sub a b

let rec mylog n = if n=1 then 0 else 1+(mylog (n/2))

let build_range_mmin a n mmin =
  (* construct an array of arrays m.().()
     where m.(i).(j) is the mmin of the 2^i elements from a.(j) ... a.(j+2^i-1)
  *)
  let k = mylog n in  (* 2^k is the biggest block we use, so use 2^0...2^k *)

  let m = Array.make (k+1) [||] in
    m.(0) <- Array.copy a;
    for kk = 1 to k do
      let b = 1 lsl kk in (* block size *)
        m.(kk) <- Array.make (n-b+1) 0;
        for j=0 to n-b do
          m.(kk).(j) <- mmin m.(kk-1).(j) m.(kk-1).(j+b/2)
        done
    done;
    (m,k,n)

let range_mmin a b (m,_,_) mmin =
  let k = mylog (b-a+1) in
  let bl = 1 lsl k in (* block size *)
    mmin m.(k).(a) m.(k).(b-bl+1)
```

```
let solve lo hi n =
  let rec expand () =
    let lo_min = build_range_mmin lo n min in
    let hi_max = build_range_mmin hi n max in
    let rec loop i ac = if i=n then ac else (
      let nlo = range_mmin lo.(i) hi.(i) lo_min min in
      let nhi = range_mmin lo.(i) hi.(i) hi_max max in
      let ac = ac || nlo <> lo.(i) || nhi <> hi.(i) in
        lo.(i) <- nlo;
        hi.(i) <- nhi;
        loop (i+1) ac
    )
    in
      if (loop 0 false) then expand () else ()
  in
    expand ();
    Array.init n (fun i -> hi.(i) - lo.(i) + 1)

let rec main() =
  let n = read_int() in
    if n=0 then () else
      let p = Array.init n (fun i -> let x = read_long() in (x, read_long(), i)) in
      let () = Array.sort compare p in
      let lo = Array.make n 0 in
      let hi = Array.make n 0 in
      let rec rightmost a b rb =
        (* find the rightmost index within [a,b] within i's radius *)
        if a=b then a else
          let m = (a+b+1)/2 in (* ceil (a+b)/2 *)
          let (xm,_,_) = p.(m) in
            if rb >= xm then rightmost m b rb else rightmost a (m-1) rb
      in
      let rec leftmost a b lb =
        if a=b then a else
          let m = (a+b)/2 in
          let (xm,_,_) = p.(m) in
            if lb <= xm then leftmost a m lb else leftmost (m+1) b lb
      in

        for i=0 to n-1 do
          let (x,r,_) = p.(i) in
            lo.(i) <- leftmost 0 (n-1) (x--r);
            hi.(i) <- rightmost 0 (n-1) (x++r);
        done;

        let ans = solve lo hi n in
        let oans = Array.make n 0 in
          for i=0 to n-1 do
            let (_,_,j) = p.(i) in
              oans.(j) <- ans.(i)
          done;
          for j=0 to n-1 do
            if j>0 then Printf.printf " ";
            Printf.printf "%d" oans.(j)
          done;
          print_newline();
          main ()
;;
main()
```