IOI Training Camp, Lecture Notes, Day 1

Fri 20 Jun 2008

Session 1, 11:00--12:30

Ramu's mango trees

Ramu's father has left a farm organized as an N x N grid. Each
square in the grid either has or does not have a mango tree. He
has to divide the farm with his three sisters as follows: he will
draw one horizontal line and one vertical line to divide the
field into four rectangles. His sisters will choose three of the
four smaller fields and he gets the last one.

He wants to divide the field so that he gets the maximum number
of mangos, assuming that his sisters will pick the best three
rectangles.

For example, if the field looks as follows,

```
.##...
#..##.
.#....
.##..#
#..##.
.#....
```

Ramu can ensure that he gets 3 mango trees by cutting as follows

```
.#|#...
#.|.##.
.#|....
------+---------
.#|#..#
#.|.##.
.#|....
```

Solution:

If Ramu draws the vertical line after column x and the horizontal line after row y, we represent it by (x,y). For each cut (x,y), we need to calculate the minimum rectangle that it creates. Then, over all the cuts, we choose the one whose minimum rectangle is maximized.

The problem is to efficiently calculate how a cut divides up the mango trees in the four rectangle.s

Let M[x,y] be the number of mango trees in the rectangle between (0,0) and (x,y)

We can calculate M[x,y] as follows:

$$M[x,y] = 1 + M[x-1,y] + M[x,y-1] - M[x-1,y-1], \text{ if tree at } (x,y)$$
$$= 0 + M[x-1,y] + M[x,y-1] - M[x-1,y-1], \text{ if no tree at } (x,y)$$

Thus, M[x,y] is the number of mango trees in the top left rectangle formed by the cut. Using M[x,y], we can also calculate the number of mango trees in the top right and bottom left rectangles defined by (x,y).

Top right rectangle:

M[N,y] - M[x,y] : Number of mangos in rectangle
defined by (x,y) and (N,0)

Bottom left rectangle:

M[x,N] - M[x,y] : Number of mangos in rectangle
defined by (0,N) and (x,y)

Finally, we Subtract these three quantities from total number of mango trees to get number of mangos in fourth rectangle (bottom right), defined by (x,y) and (N,N).

==
=======
Maximum sum subsection

Given a sequence of integers, a subsection is a block of
contiguous integers in the sequence. The task is to find the
subsection whose sum is maximum.

For instance, here is a sequence and a subsection marked out
whose sum is -1

```
  3 -1  2 -1  2 -3  4 -2  1  2  1
   |<------------>|
      sum = -1
```

Naive solution:

Let the sequence be A[1], A[2], ... A[N]
Pick each possible pair of positions (i,j) and compute the sum of
the subsection from A[i] to A[j].

There are N^2 such choices for (i,j) and we need to scan the
length of the subsection, which requires work proportional to N
in general. Overall, this takes time N^3.

A smarter solution:

For each position i, compute the prefix sum A[1] + A[2] + ... +
A[i]. Let PrefSum[i] denote this value. We can compute
PrefSum[i] for each i in a single scan.

Now, for each pair (i,j), we get the sum of this subsection as

   PrefSum[j] - PrefSum[i]

which takes constant time.

Thus, the overall problem comes down to complexity N^2 from N^3.

Still smarter:

This problem can actually be solved in time N. How?   dp :p

==
=======

Buying Empty Plots

Along one side of a road there is a sequence of vacant plots of
land. Each plot has a different area (non-zero). So, the areas
form a sequence A[1], A[2], ... A[N].

You want to buy K acres of land to build a house. You want to
find a segment of continguous plots (i.e., a subsection in the
sequence) of minimum length whose sum is exactly K.

First attempt:

Can compute prefix sums as before and, for each (i,j) check if
A[j] - A[i] = K and, among all such sections, choose the one
where j - i is minimized.

This takes time N^2. Can we do better?

The prefix sums form a sequence of values in ascending value.

   P[1] < P[2] < ... < P[N]

For each i, we want to find a j such that P[j] - P[i] = K. Since
the prefix sums are sorted in ascending order, we can find the j
for each i using binary search, instead of exhaustively scanning
P[i+1], ..., P[N].

This improves the complexity to N log N.

Can we do it in time proportional to N?

For each i, compute if there is a segment starting at i that
adds up to exactly K.

Start at A[1]. Add A[2], A[3] ... .

- If we reach A[j] such that sum till A[j] is exactly K, note
  this down.

- If the sum till A[j-1] is less than K and the sum till A[j] exceeds
  K, there is no solution starting from A[1].

At this point, we subtract A[1] from the sum. We automatically have the sum starting from A[2] upto A[j].

The sum from A[2] upto A[j-1] must be less than K (since the sum from A[1] to A[j-1] was less than K).
If there is already a solution for
A[2], it must be A[2] to A[j].

- If A[2] + .. + A[j] is exactly K, note this down and subtract A[2] to get the sum from A[3].

Otherwise, we continue to add A[j+1] etc until we reach a value where the sum crosses K and repeat the argument above.

Essentially we maintain a sliding window within which we maintain the sum. At each stage

- if the sum of the window is less than K, expand the window to the right.

- if the sum of the window is greater to K, contract the window from the left.

- if the sum of the window is equal to K, note the left and right end points and contract the window from the left.

Example:

  K = 8

  1   3   2   1   4   1   3   2   1   1   2

  Expand window right till sum crosses K
  |--->4--->6--->7-->11

  Exceeds K, contract window from left
      |------------>10

  Exceeds K, contract window from left
          |------->7

  Less than K, expand window to right

```
       |------------>8
```

Equals K, note left and right end points, contract from left
```
         |-------->6
```

Less than K, expand window to right
```
         |------------->9
```

Exceeds K, contract window from left
```
           |-------->8
```

Equals K, note left and right end points, contract from left
```
              |--->4
```

Less than K, expand window to right
```
              |--->4--->6--->7--->8
```

Equals K, note left and right end points, contract from left
```
                 |------------->7
```

Less than K, expand window to right
```
                 |------------->7--->9
```

Nice problem and good idea :)

Exceeds K, contract window from left
```
                   |------------->7
```

Less than K, cannot expand window to right, stop

===
======
Buying TWO Empty Plots

We have a sequence of plots as before, but we want to build two
buildings requiring K acres each, so we want to buy two such
subsections so that the number of plots in the two subsections is
minimized.

Note that the two subsections we require must be disjoint, so it
is not enough to pick the two best subsections overall.

Naive solution:

Find all possible K length subsections. We have to find a pair
that does not overlap with minimum combined length. Check this
property for all possible pairs. Can we avoid checking all
pairs?

Hint:

For each position i, find the best K length segment to the left
and right of i using the solution to the previous problem for
A[1] A[2] ... A[i] and A[i+1] A[i+2] ... A[n].

Call these segments LBEST[i] and RBEST[i] (best K length segments
to left and right of i, respectively).

```
   1 ........................ i ......................... N
       |<--LBEST[i]--->|       |<---RBEST[i]--->|
```

Naively it will take time N to calculate LBEST[i] and RBEST[i]
for each position i, so overall it will take time N^2.

A better approach:

For each i, compute

  lbest[i] : Best K-length sequence beginning at i
  rbest[i] : Best K-length sequence starting at i

Note the difference between lbest/rbest and LBEST/RBEST. For
lbest/rbest, the segment must end/start at i, respectively.

```
   1 ........................ i ......................... N
          |<--lbest[i]--->|<---rbest[i]--->|
```

The solution to the previous problem finds rbest[i] for each i
(i.e. sequence of K starting at i).

We reverse the computation from right to left to get lbest[i].

So, we can compute rbest[i] and lbest[i] for each i in time
proportional to N.

Now,

```
RBEST(i) = min rbest(j) for j >= i
LBEST(i) = min rbest(j) for j <= i
```

Each of these can be computed in a single scan. For instance, as
we vary i from 1 to N, we keep the minimum value of lbest[i] seen
so far and update it as LBEST[i] for each i.

===
=======

Garden (IOI 2005, Poland)

A garden is arranged as a rectangular grid of squares. Each
square contains a number of roses. We want to hire two gardeners
to take care of the roses. Each gardener is expected take care
of exactly K roses, where K is a fixed number. Each gardener
wants to work on a rectangular patch. To avoid the gardeners
interfering with each other, they should work on with both
patches disjoint (though it is permissible for the two rectangles
to share a common boundary). Each rectangular patch comes with a
cost, which is the perimeter of the patch. The aim is to find
two non overlapping rectanglular patches, each containing exactly
K roses, so that the total cost of the two patches is minimal.
(If the two rectangles share a portion of the boundary, this is
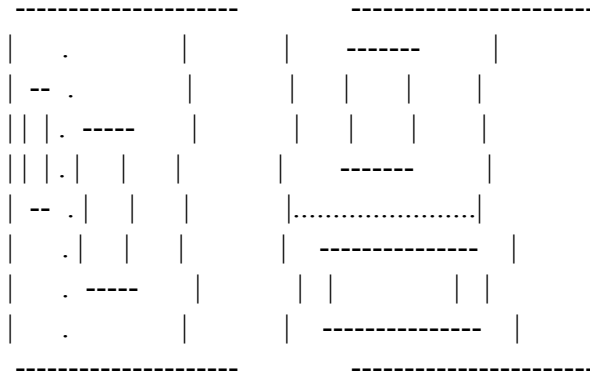to be counted in the perimeter of both rectangles).

Here the number of rows and columns is <= 250, number of roses
<= 5000, K <= 2500.

Solution

It is not sufficient to greedily find one rectangle of minimum
perimeter and then search for the second one. In the
minimum cost pair, neither rectangle may be of minimum size
overall.

Observation:

If we take two non-overlapping rectangles, they can always be
separated by a vertical or horizontal line.

```
   ---------------------            -----------------------
  |    .            |         |       -------         |
  | -- .            |         |      |   |    |       |
  || |. -----       |         |      |   |    |       |
  || |.|   |   |    |         |       -------         |
  | -- .|   |   |            |.........................|
  |    .|   |   |            |    --------------     |
  |    . -----       |         | |              | |
  |    .            |         |    --------------     |
   ---------------------            -----------------------
```
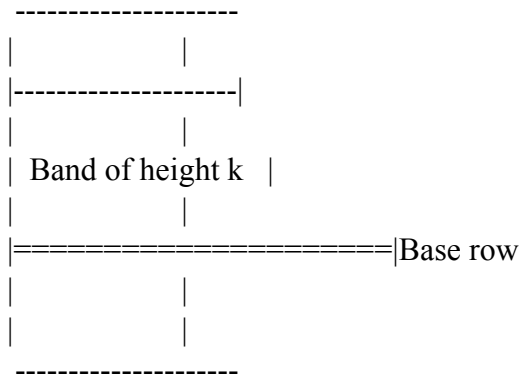
Thus, we can try all possible ways of splitting the original
rectangle using either horizontal or vertical line. For each
split, we have to find the single best rectangle in each part.

In this way, we have reduced the problem from jointly identifying
two rectangles with minimum cost to that of finding one minimum
cost rectangle within a given rectangle.

How can we systematically enumerate all rectangles?

Assume we are using fixed height rectangles. Systematically
choose each row of the garden to be the "base row".   For each choice of
base row, check each band that uses this base row.
Symmetrically, fix a base column for vertical bands.

```
    ---------------------
   |           |
   |--------------------|
   |           |
   | Band of height k   |
   |           |
   |====================|Base row
   |           |
   |           |
    ---------------------
```

This still requires us to calculate the number of roses in a
given rectangle. How can we do this efficiently without actually
adding up all values within the rectangle?

Calculate number of roses for special kind of rectangles, which
start at top left corner of the original rectangle.

```
   1 2 3 4 5 6 7 8 9  ... m
  ------------------------------------
1 | | | | | | | | | | | |...| |
  ------------------------------------
2 | | | | | | | | | | | |...| |
  ------------------------------------
3 | | | | | | | | | | | |...| |
  ------------------------------------
. | | | | | | | | | | | |...| |
.  ------------------------------------
. | | | | | | | | | | | |...| |
  ------------------------------------
n | | | | | | | | | | | | | |
  ------------------------------------
```

Count(i,j) is the number of roses in the rectangle whose top left corner is (1,1) and bottom right corner is (i,j). We can compute Count(i,j) if we know the neighbouring values:

```
   ------------------------------
   |        .      .        |
   |        .      .        |
   |        .      .        |
   |......(i-1,j-1) ----- (i-1,j)  |
   |         |       |        |
   |         |       |        |
   |....... (i,j-1) ----- (i,j)    |
   |                        |
   ------------------------------
```
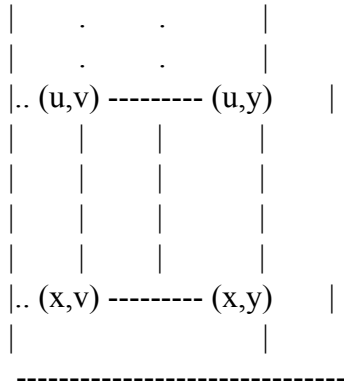
Count(i,j) = Count(i-1,j) + Count(i,j-1) - Count(i-1,j-1) + Roses(i,j)

where Roses(i,j) is number of roses at (i,j).
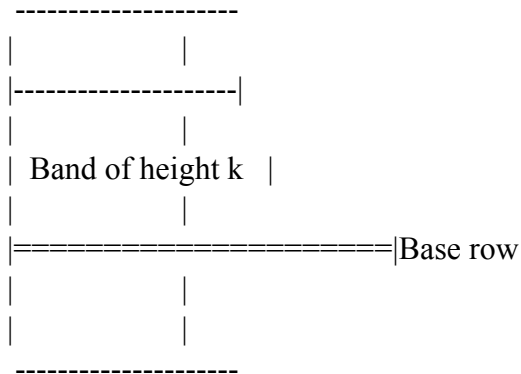
Now, given a rectangle with top left corner (u,v) and bottom right corner (x,y), the count of this rectangle is:

Count(x,y) - Count(u,y) - Count(x,v) + Count(u,v)

```
   ------------------------------
   |      .      .        |
```

```
|    .    .       |
|    .    .       |
|.. (u,v) --------- (u,y)      |
|    |    |        |
|    |    |        |
|    |    |        |
|    |    |        |
|.. (x,v) --------- (x,y)      |
|                 |
     -------------------------------
```
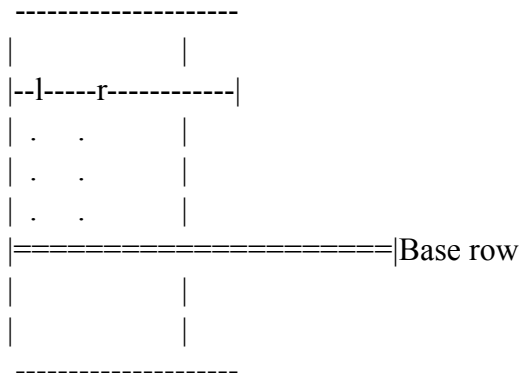
Now, sitting on base row i we want to compute the best K-rose rectangle whose bottom edge is on i and whose top edge is on row i-k.

```
     ---------------------
    |               |
    |---------------------|
    |               |
    |  Band of height k   |
    |               |
    |====================|Base row
    |               |
    |               |
     ---------------------
```

For this, we use the sliding window technique from the Empty Plot problem.

```
     ---------------------
    |               |
    |--l-----r------------|
    | .    .        |
    | .    .        |
    | .    .        |
    |====================|Base row
    |               |
    |               |
     ---------------------
```

Given l and r, we can find the number of roses in this rectangle, using the counts we have calculated. This is really a one dimensional sliding window problem. If the count in the

rectangle defined by l and r is less than or greater than K we expand/contract the window as before. If the count is exactly K, we compute the perimeter and note it down.

In this way, we can compute the best rectangle sitting on base row i as the minimum among all best rectangles whose top row ranges from 1 to k-1. For each top row, the sliding window takes time N. There are M rows above i, so this takes MN time.

Similarly, we can find all rectangles whose top row is the base row i and bottom row varies from i+1,...,M.

We do this for every base row, so this takes time M times MN = $M^2 N$ overall.

In a similar way, we fix base columns and find the best rectangles to the left and right. This takes N times MN = $N^2 M$ overall.

==
=======
Session 2, 14:00--15:30
==
=======
A simple tiling problem

We have an (n x 2) grid to be tiled.

```
<-------- n --------->
 -- -- -- -- -- -- --
| | | | | | |..| |
 -- -- -- -- --   --
| | | | | | |..| |
 -- -- -- -- -- -- --
```

We have with us a supply of rectangular tiles of   -- --
size 2 x 1. Each tiles can be rotated and laid | | |
horizontally or vertically.   -- --

How many ways can we tile the (n x 2) grid using these tiles?

Solution

Let T(n) denote the number of ways of tiling an (n x 2) grid.

We begin the tiling at the left end. We have two possibilities:

(a) We place a tile vertically. This leaves an ((n-1) x 2)
    grid, that can be tiled in T(n-1) ways

(b) We place a tile horizontally. Then, to complete the tiling,
    we are forced to place another tile horizontally below (or
    above) it. This leaves an ((n-2) x 2) grid, that can be
    tiled in T(n-2) ways.

Thus, we can express T(n) using the recurrence:

   T(n) = T(n-1) + T(n-2)

For the base case we have

   T(0) = 1 (if we have no grid, there is only one way to tile it!)
   T(1) = 1 (we must place a single tile vertically)

If you are not happy with T(0) as a base case, you can add

   T(2) = 2 (place two tiles horizontally, or two tiles vertically)

Then, we can compute T(k) for any k starting at the base case

   T(3) = T(2) + T(1) = 2 + 1 = 3
   T(4) = T(3) + T(2) = 3 + 2 = 5
   ..

This yields the familiar Fibonacci sequence: T(n) is the nth
Fibonacci number.

How do we compute T(n)?

Suppose we compute T(8) recursively. This generates subproblems as
follows:

                T(8)

```
          /        \
      T(7)          T(6)
      / \          /   \
    T(6)    T(5)  T(5)    T(4)
    / \  / \  / \    / \
 T(5) T(4) T(4) T(3 T(4) T(3) T(3 T(2)
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .
```
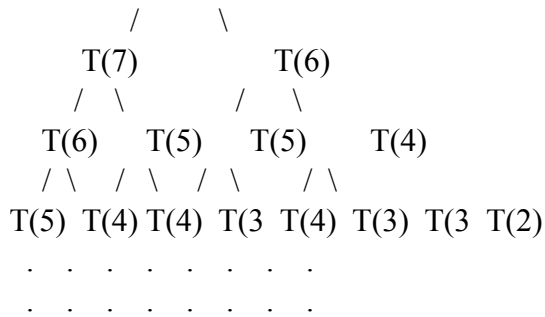
Observe that T(6) is computed twice, T(5) is computed 3 times, T(4) is computed 5 times, ...

We can modify the recursive procedure to store values that are already computed and avoid recomputation.

```
  Initialize S[1..N] to -1   // -1 denotes that S[i] is yet to
                             //  be computed

  T(i) {
    if (S[i] != -1){        // S[i] has been computed
      return(S[i])
    }else{                  // Compute T[i] and store in S[i]
      if (i == 1){
        S[i] = 1;
        return(1);
      }
      if (i == 2){
        S[i] = 1;
        return(1);
      }
      if (i > 2){
        S[i] = T(i-1) + T(i-2);
        return(S[i]);
      }
    }
  }
```

This will compute T(n) in time proportional to n, but there are other overheads associated with recursive calls. There is also a limit to the depth of recursion --- how large an n you can use for nested recursive calls.

A more efficient solution is to start with T(1) and T(2) and

work upwards to n.

```
T(i) {
  if (i <= 2){
    return(1);
  }
  if (i > 2){
    S[1] = 1;
    S[2] = 1;
    for (j = 2; j <= i; j++){
      S[j] = S[j-1] + S[j-2];
    }
    return(S[j]);
  }
}
```

What if we want T(n) for very large n, for instance n = 10^8?
The solution we just wrote requires us to store 10^8 values in
the array S. Note, however, that we only need the previous two
values in the array, so we can make do by keeping two values at a
time.

```
T(i) {
  if (i <= 2){
    return(1);
  }
  if (i > 2){
    secondlast = 1;          // T(i-2)
    last = 1;                // T(i-1)
    for (j = 2; j <= i; j++){
      new = last + secondlast;  // T(i) = T(i-1) --> T(i-2)
      secondlast = last;       // T(i-1) --> T(i-2)
      last = new;              // T(i)   --> T(i-1)
    }
    return(last);
  }
}
```

==
=======

A more complicated tiling problem

Again we want to tile an (n x 2) grid, but we have two types of
tiles:

(1) A 2 x 1 tile as before   -- --
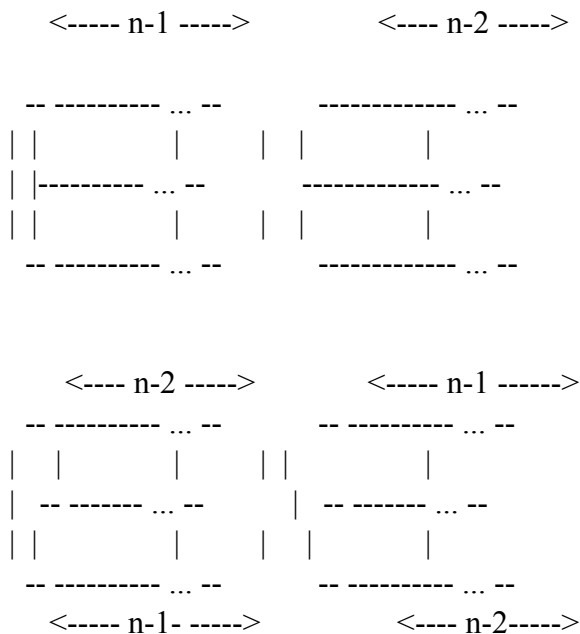 | | |
                                  -- --

(2) An L-shaped tile covering 3 squares   -- --
                                            | | |
                                             -- --
                                            | |
                                             --

How many ways can we tile the (n x 2) grid using these tiles?

Solution:

As before, let us see how we can begin our tilings from the left
end. There are now four possibilities for the leftmost tile(s):


```
    <----- n-1 ----->              <---- n-2 ----->


   -- ---------- ... --        ------------- ... --
  | |           |     | |            |
  | |---------- ... --        ------------- ... --
  | |           |     | |            |
   -- ---------- ... --        ------------- ... --



    <---- n-2 ----->              <----- n-1 ------>
   -- ---------- ... --        -- ---------- ... --
  | |    |      |    | |             |
  |  -- ------- ... --        |  -- ------- ... --
  | |           |     | |            |
   -- ---------- ... --        -- ---------- ... --
    <----- n-1- ----->             <---- n-2----->
```


If we use an L-shaped tile initially, we are left with a
shape in which either the top or bottom row has one extra

square.

Let

- f(n) = number of ways of tiling an (n x 2) rectangle.
- g(n) = number of ways of tiling an (n x 2) rectangle
    with an extra square in the bottom row.
- h(n) = number of ways of tiling an (n x 2) rectangle
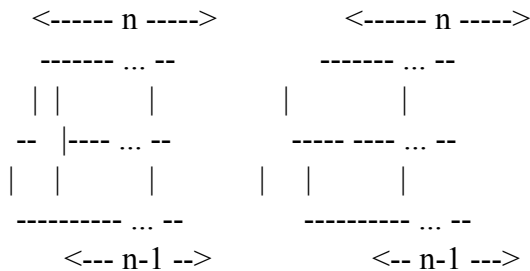   with an extra square in the top row.

From cases above, we can see that

$$f(n) = f(n-1) + f(n-2) + g(n-2) + h(n-2)$$

Observe that, by symmetry, h(n) = g(n), so we can simplify this to

$$f(n) = f(n-1) + f(n-2) + 2g(n-2)$$

In a similar fashion, we can write a recurrence for g(n) by
looking at what we can do initially.

```
    <------ n ----->           <------ n ----->
     ------- ... --             ------- ... --
     | |       |               |          |
    -- |---- ... --            ----- ---- ... --
     | |       |             | |        |
     ---------- ... --         ---------- ... --
      <--- n-1 -->              <-- n-1 --->
```

g(n) = f(n-1) + h(n-1)
    = f(n-1) + g(n-1), since g(n) = h(n)

We can write down base cases, as before:

f(0) = 1
f(1) = 1

g(0) = 1
g(1) = 1 (use exactly one L-shaped tile)

To calculate this, we start with the base case and work upwards to f(k) or g(k) for any k, as in the previous example.

Again, we should NOT use naive recursion, because this will involve wasteful recomputation of smaller values.

===
=======
Counting paths in a grid

You have a rectangular grid of points with n rows and n columns. You start at the bottom left corner. At each step, you can either go up to the next point in the same column or right to the next point in the same row. How many such paths are there from the bottom left corner to the top right corner?

Solution:

We have to take 2n steps, of which we have to choose n to be up moves and n to be right moves. This gives (2n choose n) paths.

Instead, a recursive formulation:

  Path(i,j) : No of ways of going from origin to (i,j)

   Path(i,j) =   Path(i-1,j)
          + Path(i,j-1)

  i.e. Any path to (i,j) comes via (i-1,j) or (i,j-1) and no
       path can come via both these routes, so we can
       recursively solve the problems for these two predecessor
       points and add them.

  Boundary conditions:

    Path(i,0) = 1
    Path(0,j) = 1

  This is a recurrence relation. Solving this naively is
  inefficient, because Path(i,j) will be recomputed several

times.

Instead, start at bottom left corner and compute the values along diagonals with points (i,j) where i+j is constant. Write a loop to store these values in an array. Takes time proportional to n*n (each grid point is updated exactly once and read upto twice).

One way to structure the solution as a simple nested loop is:

```
   for c = 0 to n*n
     for each (i,j) such that i+j = c
       update Path(i,j) using Path(i-1,j) and Path(i,j-1)
```

This way, whenever a new value Path(i,j) has to be computed, the values it depends on --- Path(i-1,j) and Path(i,j-1) --- are already known.

Why is the recursive solution better?

What if some points are deleted (that is, no path can go through these points)?

If there is one deleted point, we have to subtract the number of paths that pass through these points.

In general, holes may be arbitrarily complicated: if we have two holes, we subtract number of paths that pass through either hole, but we have then subtracted twice the paths that go through both holes, so we have to add these back.

Instead, we can modify the recurrence to take into account the holes.

```
  Path(i,j) =   Path(i-1,j), if (i-1,j) connected to (i,j)
              + Path(i,j-1), if (i,j-1) connected to (i,j)

  Path(i,0) = 0 or 1, depending on whether there is a hole
                 between (0,0) and (i,0)
  Path(0,j) = 0 or 1, depending on whether there is a hole
                 between (0,0) and (0,i)
```

===

=======

Longest common subword

Suppose we have two words over some letters, such as

 V = T G A T G G A T C
 W = G T T T G C A C

We want to find the longest contiguous sequence of letters
(subword) that is contained in both words. In this case, the
longest common subword is "T G".

  LCSW[i,j] = length of longest common subword starting at
         position i in V and position j in W

  LCSW[i,j] = 0, if V[i] != W[j]
        = 1 + LCSW[i+1,j+1], otherwise

  Base case:

  LCSW[M,j] = 1 if V[N] = W[j], 0 otherwise
  LCSW[i,N] = 1 if V[i] = W[N], 0 otherwise

  We calculate B[i,j] in an M x N array. We know the values in
  the last row and last column from the base case. Once we know
  the value at (i+1,j+1), we can compute the value at (i,j). So,
  we can systematically fill up the values for B[i,j] row by row,
  right to left, starting from the bottom (or, equivalently,
  column by column, bottom to top, starting from the right).

==
=======

Longest common subsequence

As before, we have two words over some letters, such as

 V = T G A T G G A T C
 W = G T T T G C A C

Now we can drop some positions and obtain a subsequence. For
instance "T G A C" is a subsequence of both words,

LCSS[i,j] = length of longest common subsequence in the words
V[i] V[i+1] ... V[M] and W[j] W[j+1] ... W[N]

LCSS[i,j] = LCSS[i+1,j+1] + 1,   if V[i] = W[j]
= max (LCSS[i+1,j], LCSS[i,j+1]), otherwise

Once again, we can fill up the values LCSS[i,j] in an M x N
array, starting with the last row and column as base cases.

LCSS[M,j] = 1 if V[M] appears anywhere in W[j] ... W[N]

Fill the last row from right to left and once we fill a 1, all
entries to the left are 1.

Similarly

LCSS[j,N] = 1 if W[N] appears anywhere in V[j] ... V[N]

Fill the last column from bottom to top and once we fill a 1,
all entries above are 1.

===
=======

Exercise:

The recurrences for LCSW and LCSS only tell us the lengths of the
longest common subword and subsequence, respectively. How do we
recover a witness --- that is, a common subword or subsequence of
this maximum length?

For subword it is easy --- since we know the position where the
longest common subword starts, we can read off the subword.

How do we do it for longest common subsequence?

===
=======

Longest ascending subsequence

Let s = a[1] a[2] ... a[n] be an unsorted sequence of numbers.

A subsequence of s is one in which we drop some elements and retain the rest (so we need not pick up a contiguous segment, but we preserve the order of elements).

For instance, a[1] a[3] a[n] is a subsequence.

Problem:

Find the (length of the) longest ascending subsequence.

Solution 1:

  Sort the array and find the longest common subsequence between sorted sequence and original sequence.

  This takes time $O(n^2)$ (sorting takes time $O(n \log n)$ and longest common subsequence takes $O(n^2)$).

Solution 2:

  A direct dynamic programming solution for this problem.

  LAS[i] : length of longest ascending subsequence in that ends with a[i] (and hence only contains elements from a[1] a[2] ... a[i])

  LAS[1]   = 1

  LAS[i+1] = 1 + max { LAS[j] | 1 <= j <= i, a[j] <= a[i+1] }

    i.e. to find the length of the longest ascending subsequence ending at a[i+1], look at all values to the left of a[i+1] that are less than or equal to a[i+1] and pick the position among these with the maximum LAS value.

  This can be computed in an array. This takes time $O(n^2)$ --- for each LAS[i], we have to scan values from LAS[1] to LAS[i-1].

This solution is no better than our initial solution of sorting
the sequence and then computing the longest common subsequence.

Can we do better?

We can improve the second solution to solve the problem in O(n
log n).

Maintain an additional array M

  M[j] = of all the longest ascending subsequences of length j,
       M[i] is the smallest final value

  M[j] = min { a[k] | 1 <= k <= j, LAS[k] = j }

  Observe that M[j]'s always form an ascending sequence.

  So, we can find M[j-1] <= a[i] < M[j] using binary search

  Adding a[i] requires log(i) time

  Overall we make n updates, so the overall algorithm is n log n

An example:

i   1   2   3   4   5   6   7   8

a[i] 8   2   7   3   9   4   6   5

LAS[i] 1   1   2   2   3   3   4   4

M[i]   8X 7X 9X 6X
     2   3   4   5

==
=======
==
=======
        IOI Training Camp, Lecture Notes, Day 2

                Sat 21 Jun 2008
==

The Great Escape (IARCS Online Contest, Oct 2004)

Heroes in Tamil movies are capable of superhuman feats. For
example, they can jump between buildings, jump onto and from
running trains, catch bullets with their hands and teeth and so
on. A perceptive follower of such movies would have noticed that
there are limits to what even the superheroes can do. For
example, if the hero could directly jump to his ultimate
destination, that would reduce the action sequence to nothing and
thus make the movie quite boring. So he typically labours through
a series of superhuman steps to reach his ultimate destination.

In this problem, our hero has to save his wife / mother / child /
dog /... held captive by the nasty villain on the top floor of a
tall building in the centre of Bombay / Bangkok / Kuala Lumpur
/.... Our hero is on top of a (different) building. In order to
make the action "interesting" the director has decided that the
hero can only jump between buildings whose height is within 10
metres.

Given the arrangement of buildings and their heights, your aim is
determine whether it is possible for the hero to reach the
captive.

Here is an example where the goal is to get from the top left
building to the bottom right building.

        70   80   80   70

        60   30   20   30

        50   40   10   50

        45   60   20   10

Here is a possible path:

        70   80   80   70

```
     |
   60   30 -- 20   30
   |    |    |
   50 -- 40   10   50
              |
   45   60   20 -- 10
```

How do we find such a path? We systematically calculate all the buildings that are reachable. For each building, we know (by the heights) which of the neighbouring buildings can be reached in one step.

-- Initially, the starting position (1,1) is reachable

-- If (i,j) is reachable, (i',j') is a neighbour of (i,j) and the
   height difference is at most 10, then (i',j') is also
   reachable.

Using this procedure, we can grow the set of all reachable buildings until we cannot add any more buildings.   At this point, we check if the target building is in the reachable set.

How do we code this?

A recursive solution:

```
  Building heights are stored in a matrix A[N][N]
  Currently reachable buildings are in a matrix Mark[N][N],
    initialized to 0

  check(i,j){ // Assuming that (i,j) has just been marked, n
          // mark all its neighbours
    if ((i-1,j) within grid &&           // Nbr within grid
        abs(A[i-1][j] - A[i][j] <= 10) &&  // Difference is OK
        Mark[i-1][j] == 0){              // Nbr unmarked

        Mark[i-1][j] = 1;
        check(i-1,j);
    }

    /* Similarly check (i+1,j), (i,j-1), (i,j+1) */

    ...
```

```
 }
```

For each (i,j), check(i,j) is called at most once and processing
check(i,j) takes constant time since each building has at most
four neighbours, so this will take time proportional to N^2, the
total number of buildings.

Shortest route

What if, in addition, we want to compute the shortest route from
(1,1) to (N,N)?

An efficient way to maintain and process marked positions is
to use a queue (first-in-first-out collection).

  Initially, mark the starting position and add it to the queue.

  In each round:

  (Assume we have a queue of marked positions whose neighbours
  are yet to be explored.)

  Extract the first element from the queue. Mark all unmarked
  reachable neighbours (whose height difference is within the
  bound) and insert them into the queue.

  Repeat this till the queue becomes empty (there are no more
  marked positions to examine).

For instance:

        70   80   80   70

        60   30   20   30

        50   40   10   50

        45   60   20   10

Mark (1,1) and start with Queue = {(1,1)}

Extract (1,1) from head of queue, mark its reachable neighbours
and add them to the Queue: Queue is now {(1,2),(2,1)}

Extract (1,2) from head of queue, mark its reachable neighbours
and add them to the Queue: Queue is now {(2,1),(1,3)}

Extract (2,1) from head of queue, mark its reachable neighbours
and add them to the Queue: Queue is now {(1,3),(3,1)}

Extract (1,3) from head of queue, mark its reachable neighbours
and add them to the Queue: Queue is now {(3,1),(1,4)}

Extract (3,1) from head of queue, mark its reachable neighbours
and add them to the Queue: Queue is now {(1,4),(3,2),(4,1)}

...

Let us examine the order in which positions get marked. When we
explore a position that is k steps away from the starting
position, its neighbours are k+1 steps away and get added later
in the queue. By maintaining the marked positions in the queue,
we guarantee that positions are marked in increasing order of
their distance from the source. We can also record the distance
with each node as we visit it.

This method of exploration, in which we explore all neighbours
level by level, is called breadth first search.

The earlier recursive formulation (using "check(i,j)") is depth
first search. We pick a neigbhour to explore, then explore its
neighbours, ... till we can make no progress, and then we come
back to process the next unmarked neighbour of the first node.

Here is a more precise formulation of breadth first search:

  Mark[(i,j)] = 0 for all (i,j)

  Mark[(1,1)] = 1;
  Dist[(1,1)] = 0;
  Insert (1,1) into the queue;

```
  BFS {
    while (queue is not empty) {
      v = head of queue;
      for each unmarked neighbour w of v {
        Mark[w] = 1;
        Dist[w] = Dist[v] + 1;
        Insert w into the queue;
      }
    }
  }
```

From rectangular grids to the general case

Suppose the director shifts his shooting location to a venue
where buildings are no longer arranged in a nice rectangular
grid.

Assume we have N buildings 1,2,...,N and a list of pairs
{(i1,j1),(i2,j2),...,(iM,jM)} where each pair (i,j) says that
building i and building j are "neighbours" --- that is, they are
near enough in distance and height for the hero to jump from i to
j or j to i. The hero has to start at building 1 and the captive
is in building N.

For instance, suppose we have 7 buildings 1,2,...,7 and the list
of neighbours are given by
      {(1,2),(1,3),(2,3),(3,4),(5,6),(7,6),(5,7)}.
Can the hero rescue the captive?

No, because there is no way to go from the set of buildings
{1,2,3,4} to the set of buildings {5,6,7}.

If we draw this as a picture, it is easy to see that this problem
has no solution:

```
            2
           / \
          /   \
         /     \
        1 --------- 3 -------- 4
```

```
   5 --------- 6
     \     /
      \  /
       7
```
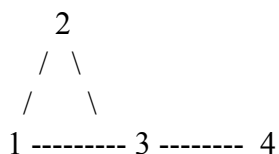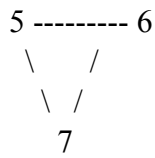
Here, each building is labelled and we draw a line from i to j if
i and j are neighbours. From the picture, it is immediate that
there is no way to travel from {1,2,3,4} to {5,6,7}.

This kind of picture is called a graph. A graph is a pair G =
(V,E) where

-- V is a set of vertices (or nodes)

    In our example, these represent the buildings

-- E is a set of pairs of positions, called edges

    In our example, these represent the pairs of buildings that
    are neighbours

How do we check reachability in this framework?

We can use a queue as before:

Mark starting position 1 and start with queue {1}.

Unmarked neighbours of 1 are {2,3}, so mark {2,3} and update queue
to {2,3}.

Neighbours of 2 are {1,2} but both are already marked, so do
nothing and update queue to {3}.

Unmarked neighbour of 3 is {4} so mark 4 and update queue to {4}.

Neighbour of 4 that is already marked, so do nothing and update
queue to {}.

Queue is empty, so stop. Marked nodes are {1,2,3,4} which are
precisely the nodes reachable from 1.

Notice that we have used exactly the same algorithm as we did for

the rectangular grid. The only difference is that instead of
implicitly identifying neighbours with respect to grid positions,
we are explicitly given the neighbours for each node.

Breadth first search for general graphs

BFS(v){

  mark(v);
  add v at the end of the Queue; /* Queue was empty */

  while the Queue is not empty{
    remove the vertex w at the head of the Queue;
    for each unmarked neighbour u of w{
      mark(u);
      add u to the end of the Queue;
    }
  }
}


```
    1              BFS(1) generates the following queue:
   /\
   2---3
   |           1, 2, 3, 4, 5, 8, 6, 7
   4---8
   | |
   5  7
   \ /
    6
```

The order which the vertices are visited by BFS gives the
shortest path. How do we recover the length of the path? We can
record the length of the shortest path to each vertex w when we
add it to the queue. This is just one more than the length of
the shortest path to its parent.

BFS(v){

  mark(v);
  length(v) = 1;

```
    add v at the end of the Queue;  /* Queue was empty */

   while the Queue is not empty{
     remove the vertex w at the head of the Queue;
     for each unmarked neighbour u of w{
       mark(u);
       length(u) = length(w)+1;
       add u to the end of the Queue;
     }
   }
}
```

To actually recover the path, once again, we can also record
parent(u)=w when we explore w and add u to the queue.

```
BFS(v){

  mark(v);
  length(v) = 1;
  add v at the end of the Queue;  /* Queue was empty */

   while the Queue is not empty{
     remove the vertex w at the head of the Queue;
     for each unmarked neighbour u of w{
       mark(u);
       length(u) = length(w)+1;
       parent(u) = w;
       add u to the end of the Queue;
     }
   }
}
```

BFS tree

When we explore a graph using breadth first tree, we can
represent the visited nodes as a tree, called the BFS tree. A
tree is a connected graph without cycles.

Paths and cycles:

   A path is a sequence of distinct vertices v_1 v_2 ... v_k

where each adjacent pair (v_i,v_{i+1}) is an edge.

A cycle is a sequence of vertices v_1 v_2 ... v_k v_1, where
v_1 v_2 ... v_k is a path and v_k v_1 is an edge. (i.e. a
cycle is a path that is closed by an edge from the last vertex
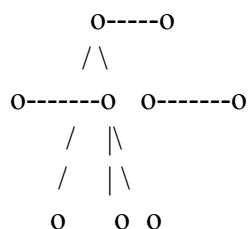in the path back to the initial vertex.)

Connected:

v_1 and v_2 are connected if there is a path from v_1 to v_2
A graph is connected if every pair of nodes is connected.
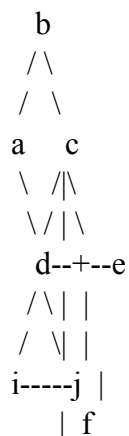
How do we check if a graph is connected?

An undirected graph is connected precisely when DFS(v) or
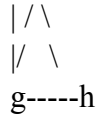BFS(v) marks all vertices (you can choose any start vertex v).

Tree:

A graph without cycles (in an undirected sense) with exactly
one component, that is, the graph is connected.

```
       o-----o
        / \
  o-------o  o-------o
     /  |\
     /  | \
     o    o o
```

BFS tree example:

```
      b
     / \
    /  \
    a    c
     \  /|\
      \/|\
       d--+--e
      / \| |
      / \| |
     i-----j |
        | f
```

```
| / \
|/   \
g-----h
```
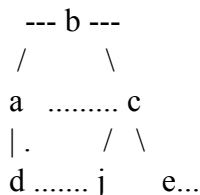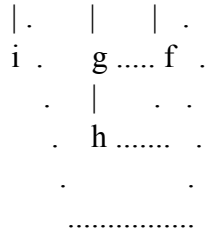
Suppose we start BFS at b

From b, we visit a,c (level 1)   --- queue is {a,c}
From a, we visit d (level 2)   --- queue is {c,d}
From c, we visit j,e (level 2)   --- queue is {d,j,e}
From d, we visit i (level 3)   --- queue is {j,e,i}
From j, we visit g (level 3)   --- queue is {e,i,g}
From e, we visit f (level 3)   --- queue is {i,g,f}
From i, no unmarked neighbours   --- queue is {g,f}
From g, we visit h (level 4)   --- queue is {f,h}
From f, no unmarked neighbours   --- queue is {h}
From h, no unmarked neighbours   --- queue is {}
Empty queue, so we stop

If draw the vertices that we visit in BFS level by level, we get
the following tree, called the BFS tree.

```
    ---b ---
    /      \
   a        c
   |       / \
   d      j   e
   |      |   |
   i      g   f
          |
          h
```
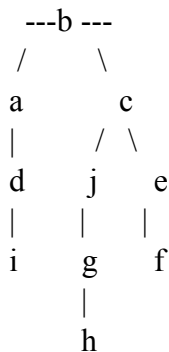
The edges in this tree come from the graph. There are some edges
in the graph that do not appear in this tree. In this case, they
are {(c,d), (d,e), (d,j), (f,g), (f,h)}. If we add these edges
to this tree we get a picture like this.

```
    --- b ---
    /       \
   a  ......... c
   | .        / \
   d ....... j   e...
```

```
     |.     |     |  .
     i  .     g ..... f   .
          .   |     .  .
          .  h ........     .
          .              .
            ................
```

In this figure, we refer to the original edges (---) as tree
edges and the new edges (..., not used in the tree) as cross
edges.

Note that cross edges either connect at same level or adjacent
levels. Cross edges cannot cross two levels. (Why?)

===
=======

Depth first search (DFS)

The other option for exploring a graph is depth first. Pick an
unexplored neighbour, pick one of its unexplored neighbours, ...,
till you can go no further. Backtrack to the previous node and
pick the next unexplored neigghbbour ...

```
 DFS(v){
   mark(v);
   for each neighbour w of v
     if (unmarked(w))  /* w is unvisited */
       DFS(w)
   }
 }
```

e.g.
```
       c1
      /|
    c2-+-c3
      \|/
       c4
```

```
 DFS(c1): mark(c1)
      DFS(c2)  : mark(c2)
            DFS(c3)  : mark(c3)
```

DFS(c4) : mark(c4)
                              all nbrs of c4 marked
                         all nbrs of c3 marked
                  all nbrs of c2 marked
          all nbrs of c1 marked

When a vertex w is marked by DFS, we know that there is a path to
w from the start vertex. How do we recover the actual path?

If we mark vertex w from v, we record that we reached w from v,
e.g. by setting parent(w) = v

```
 DFS(v){
   mark(v);
   for each neighbour w of v
     if (unmarked(w))  /* w is unvisited */
       parent(w) = v;
       DFS(w)
   }
 }
```

Tracing back via the parent relation, we can recover a path back
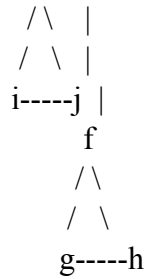from any vertex to the start vertex.

DFS numbering

Can assign a DFS number to each vertex --- order in which the
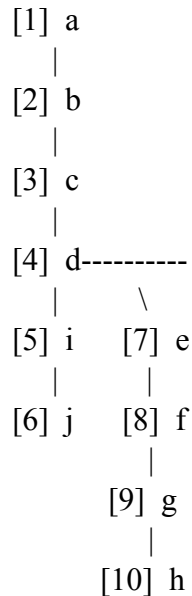vertices are explored.

Example:

        b
       /\
      / \
     a   c
      \ /\
       \/ \
        d-----e

```
        /\  |
       /  \ |
      i-----j |
            f
           /\
          /  \
         g-----h
```

DFS starting at a:

```
  [1] a
       |
  [2] b
       |
  [3] c
       |
  [4] d----------
       |       \
  [5] i     [7] e
       |         |
  [6] j     [8] f
                 |
             [9] g
                 |
            [10] h
```
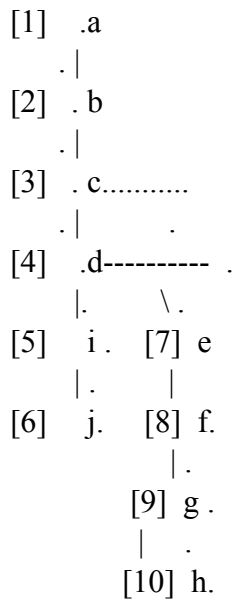
The number in square brackets, the order in which the DFS visited the vertices, is the DFS number of the vertex. (This number depends on the order in which you choose edges to explore.)

The edges that are used in the DFS form a subgraph of the original graph. This subgraph is connected and has no cycles. This is called the DFS tree of the graph. (A connected graph without cycles is called a tree.)

What about the edges in the original graph that do not form part of the DFS tree. For example, in this graph these edges are (a,d), (c,e), (f,h) and (d,j).

```
[1]   .a
       .|
[2]   . b
       .|
[3]   . c...........
       .|          .
[4]   .d---------- .
        |.      \ .
[5]    i .  [7] e
        |.      |
[6]    j.   [8] f.
                |.
            [9] g .
             |   .
           [10] h.
```

All these edges go up one of the paths in the DFS tree. The
edges cannot cross from one branch to another --- e.g., there
cannot be a non-tree edge from g to i in this DFS tree. Had
there been such an edge, we would have visited i below g, before
backtracking to e. Non tree edges are also called "back edges".

Some observations about DFS numbers of vertices:

1. DFS number of any vertex is smaller than those that appear in
   the subtree below.

1. DFS number of any vertex is smaller than those that appear to
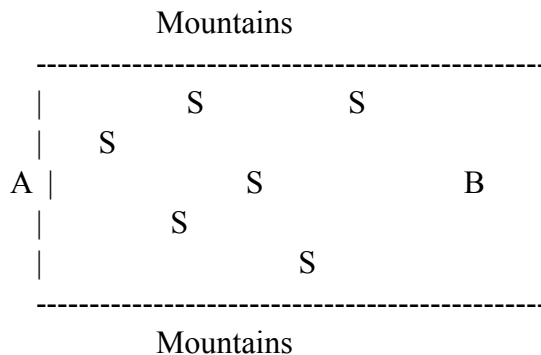   its "right" in the DFS tree.

We will use these facts later.

==
=======
Prisoners escape (Baltic Olympiad 2007)

A group of war prisoners are trying to escape from a prison. They
have thoroughly planned the escape from the prison itself, and
after that they hope to find shelter in a nearby
village. However, the village (marked as B, see picture below)
```

and the prison (marked as A) are separated by a canyon which is also guarded by soldiers (marked as S). These soldiers sit in their pickets and rarely walk; the range of view of each soldier is limited to exactly 100 meters. Thus, depending on the locations of soldiers, it may be possible to pass the canyon safely, keeping the distance to the closest soldier strictly larger than 100 meters at any moment.

```
              Mountains
    --------------------------------------------------
    |            S             S
    |      S
  A |                  S                      B
    |          S
    |                     S
    --------------------------------------------------
              Mountains
```

Given the width and the length of the canyon and the coordinates of every soldier in the canyon, and assuming that soldiers do not change their locations, you have to determine whether prisoners can pass the canyon unnoticed.

Solution

Model this as a graph. Each soldier is a vertex. Two soldiers are adjacent if they are at most distance 200 apart (there is no way to go between them).

Add a vertex s for the top boundary of the canyon and a vertex t for the bottom boundary. A soldier is adjacent to s if it is within 100 of the top border. A soldier is adjacent to t if it is within 100 of the bottom border.

Claim: If there is a sequence of guards that connect the top border to the bottom border in this graph, there is no way to escape. Otherwise, we can find a route that goes between the soldiers.

Hence, the solution just requires checking whether t is reachable from s in this graph.

Toll roads (IOI Training Camp 2006)

As the well-known saying goes, all roads lead to Siruseri. The
local government has decided to make all roads in the vicinity of
Siruseri one-way and levy a toll on each of them. The
orientation of the roads will be such that from any location
there is at least one route to Siruseri. Moreover, the tolls
will be assigned in such a way that the toll on every road is
greater than zero and, from any location, the total toll paid to
get to Siruseri is the same along any route. By route, the
government means a consecutive sequence of roads such that the
end point of each road is the starting point of the next one---a
route may pass through the same road or city more than once.

For example, suppose the roads were originally as follows, where
1 denotes Siruseri.

```
       1
      / \
     /   \
    2-------3
    |   /|
    | / |
    |/   |
    4-------5
```

We could orient the roads and assign tolls as follows:

```
      >1<
    1/   \2
    |  1  |
    2<------3
    ^  -->^
  2| |1  |2
   |/   |
    4<------5
       1
```

Here, for example, all paths from 5 to 1 have total toll 4.

Solution:

Compute DFS starting at 1. In general, this will yield a DFS
tree with back edges. Orient all edges back towards 1. Assign a
cost of 1 to all tree edges. Each backedge is a "short cut"
that bypasses a series of tree edges. Assign to the back edge a
toll equal to the sum of the tolls that it bypasses.

===
=======
Session 2, 11:00-12:30
===
=======

CMI Travel Desk

CMI travel desk insists that on booking tickets only on the
national carrier, which has an obscure pricing policy. The
cheapest way between two cities may not correspond to the most
direct route.

For instance, suppose we have cities connected by flights as
below (where 1 is CMI) with the cost of each direct flight is
shown along the edge.

```
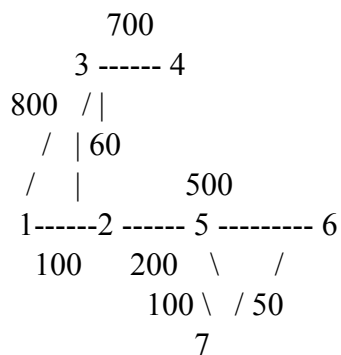             700
          3 ------ 4
     800  /|
        / | 60
       /  |          500
    1------2 ------ 5 --------- 6
      100    200  \     /
                 100 \  / 50
                     7
```

The goal is to find the cheapest route to every other city.

This is called the "Single source shortest path problem".

Here is an algorithm due to Dijkstra.

A good way to think of this problem is to imagine that there are

petrol tanks at each node, with edges corresponding to pipelines connecting the tanks whose length is given by the cost of the edge. We set fire to the first tank and the fire spreads at a uniform rate along all pipelines. If this rate is 1 unit length per second, the time at which a tank burns corresponds to the shortest path to that tank. Here is an algorithm to systematically compute the next tank that will explode at each step.

We use a modified version of BFS.

Initially, we mark the currently known distance from the start vertex to each vertex. This is 0 for start vertex, infinity everywhere else. We also mark whether a vertex is visited or not. Initially, no vertex is visited.

Initially

```
Vertex    1   2    3    4    5    6    7
Distance  0   inf  inf  inf  inf  inf  inf
Visited   0   0    0    0    0    0    0
```

Pick an unvisited vertex with the currently lowest distance to process next (this is the tank that will next explode). In this case it is 1. Visit 1 and update distances to all its neighbours.

```
Vertex    1   2    3    4    5    6    7
Distance  0   100  800  inf  inf  inf  inf
Visited   1   0    0    0    0    0    0
```

Now, 2 is the smallest unvisited vertex, so we visit 2 and update distances to all its neighbours. The distance to 3 reduces to 160 (via 2) and distance to 5 becomes 300.

```
Vertex    1   2    3    4    5    6    7
Distance  0   100  160  inf  300  inf  inf
Visited   1   1    0    0    0    0    0
```

Now, 3 is smallest unvisited vertex, so we explore it next and update distance to 4.

```
Vertex    1   2    3    4    5    6    7
```

Distance   0   100   160   860   300   inf   inf
    Visited   1   1   1   0   0   0   0

Now, 5 is smallest, so we explore it next and update distance to 6
and 7.

    Vertex   1   2   3   4   5   6   7
    Distance   0   100   160   860   300   800   400
    Visited   1   1   1   0   1   0   0

Now, 7 is smallest, so we explore it next and update distance to
6.

    Vertex   1   2   3   4   5   6   7
    Distance   0   100   160   860   300   450   400
    Visited   1   1   1   0   1   0   1

Now, 6 is smallest, so we explore it, but no further updates.

    Vertex   1   2   3   4   5   6   7
    Distance   0   100   160   860   300   450   400
    Visited   1   1   1   0   1   1   1

Finally, we visit 4.

    Vertex   1   2   3   4   5   6   7
    Distance   0   100   160   860   300   450   400
    Visited   1   1   1   1   1   1   1

At this point, we have computed shortest paths from 1 to all
other vertices.

How do we program this?

First we have to figure out how to represent a graph. One way to
do this is to use an adjacency matrix.

Adjacency matrix

Suppose we have 7 buildings 1,2,...,7 and the list of neighbours
are given by {(1,2),(1,3),(2,3),(3,4),(5,6),(7,6),(5,7)}. If we
draw this as a picture, we have the following.

```
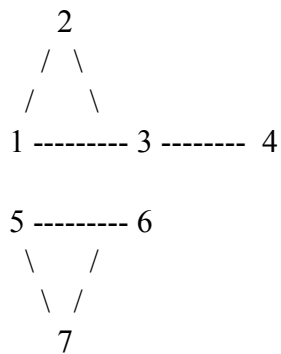            2
           / \
          /   \
     1 --------- 3 -------- 4

     5 --------- 6
       \       /
        \     /
          7
```

We can represent the structure of a graph as an "adjacency
matrix" A with 7 rows and columns (in general, n rows and n
columns, where the number of nodes in the graph is n). The entry
A[i,j] = 1 if nodes i and j are neighbours, and is 0 otherwise.

For the example above, the adjacency matrix is

```
        0 1 1 0 0 0 0
        1 0 1 0 0 0 0
        1 1 0 1 0 0 0
        0 0 1 0 0 0 0
        0 0 0 0 0 1 1
        0 0 0 0 1 0 1
        0 0 0 0 1 1 0
```

For a weighted graph, we set A[i,j] to be the weight of the edge
(i,j) rather than just 1 or 0.

Dijkstra's algorithm

Have two arrays, ExpectedBurnTime and Burnt

Initially,

  ExpectedBurnTime[1] = 0;
  ExpectedBurnTime[i] = infinity, for i other than 1
              (Use a sufficiently large value, say the
               sum of all the weights in the graph)

  Burnt[i]   = 0, for all i

Each round,

   Among all unburnt vertices, pick j with minimum
   ExpectedBurnTime. (Initially, this is 1 since Burnt[i] = 0
   for all i and only ExpectedBurnTime[1] is not infinity

   Set Burnt[j] = 1.

   Update ExpectedBurnTime[k] for each neighbour k of j, if
   necessary.

What is the complexity of Dijkstra's algorithm?

- Each round fixes the shortest path to one more vertex, so there
  are n rounds

- In each round, we pick the unmarked vertex with the smallest
  value, mark it and update values for its neighbours.

  Finding the smallest unmarked vertex takes time n

  Updating neighbours takes time n if we use an adjacency
  matrix.

So, the overall cost is n^2.

Complexity of BFS and DFS

  Let |V| = n, |E| = m.

  Note that m is at most (n choose 2), so m is O(n^2).

  We process each vertex once.

  When processing a vertex, we have to find and process all its
  neighbours. This requires traversing a row in the adjacency
  vertex (we have to pass over 0's to find 1's).

  Thus, it takes O(n^2) in the worst case.

==
=======

Representing graphs, adjacency lists

In general, assume we have N nodes and M edges. If we maintain
edges as a list, to calculate the set of neighbours of a vertex
j, we have to scan the entire list of M edges to pick out all
pair of the form (i,j) or (j,i).

What if the graph has few edges? If we do BFS with an adjacency
matrix, we cannot exploit the fact that each node has very few
neighbours.

In general, a graph has N vertices and M edges. M is at most
N(N-1)/2 or about N^2. However, M can be much smaller than N^2.
In BFS, we check each edge twice (one from each endpoint) so we
actually need only 2M steps. However, we cannot achieve this
using an adjacency matrix---with an adjacency matrix we must take
time N^2 regardless of the value M.

Adjacency list

For each vertex, maintain a list of its neighbours. The graph
with edges {(1,2),(1,3),(2,3),(3,4),(5,6),(7,6),(5,7}, is then
represented as:

     1 --> [2,3]
     2 --> [1,3]
     3 --> [1,2,4]
     4 --> [3]
     5 --> [6,7]
     6 --> [5,7]
     7 --> [5,6]

Adjacency matrix vs adjacency list

- To visit all neighbours of a node, adjacency list will be
  more efficient since we don't have to skip over 0's as we do in
  the adjacency matrix.

- To answer a question of the form "Is (i,j) an edge in the
  graph", we have to scan the list of neighbours of i, which
  could be very large. In an adjacency matrix, we just have to
  check the value of A[i,j]

So, depending on the requirement, one representation may be
better than the other.

Storing adjacency lists

Notice that a node can have at most N-1 neighbours. For
simplicity, we could represent an adjacency list as a two
dimensional array E of size [1..N] x [0..N-1]. Row i lists the
neighbours of node i. E[i][0] is the number of neighbours of i.
The entries E[i][1],...,E[i][E[i][0]] list out the actual
neighbours.

```
        0  1  2  3  4  5  6

   1)  2  2  3  x  x  x  x
   2)  2  1  3  x  x  x  x
   3)  3  1  2  4  x  x  x
   4)  1  3  x  x  x  x  x
   5)  2  6  7  x  x  x  x
   6)  2  5  7  x  x  x  x
   7)  7  5  6  x  x  x  x
```

For instance, E[6][0] is 2, saying that node 6 has two
neighbours. The actual neighbours are stored in E[6][[1] and
E[6][2].

This is not a desirable implementation of adjacency lists for two
reasons:

(a) It uses N^2 space even when M is small.
(b) If we have to initialize this matrix, it takes N^2 time.

A more efficient representation of adjacency lists

Maintain two linear arrays.

```
      Offset            Neighbours
1     1    ------------->  2
2     3    -----------      3
3     5    ----------  |->  1
4     9    --------  |      3
5     10   ------  ||--->   1
6     12   ----  ||         2
7     14   --  |||          4
            ||||            6
            ||| ----->      3
            ||  ------->    6
            ||              7
            |  --------->   5
            |               7
              ----------->  5
                            6
```

Offset[i] stores the position in Neighbours where the neighbours
of vertex i begin. To examine all neighbours of vertex i, scan
Neighbours[Offset[i]], Neighbours[Offset[i]]+1, ...
Neighbours[Offset[i+1]] - 1.

Note, however, that adding edges in this representation is
complicated.

Another way is to represent the list of neighbours using two
arrays as follows. Here, each neighbour points (via "Next") to
the next neighbour, until the last neighbour which has Next =
-1.

```
      Offset             Node  Next
1     1    ------------->  2    2
2     3    -----------      3   -1
3     5    ----------  |->  1    4
4     9    --------  |      3   -1
5     10   ------  ||--->   1    6
6     12   ----  ||         2    7
7     14   --  |||          4    8
            ||||            6    9
            ||| ----->      3   -1
            ||  ------->    6   11
            ||              7   -1
```

```
              | --------->  5   13
              |            7  -1
               ----------->  5   15
                           6  -1
```

Suppose we now add a new neighbour to 2, say 7. We append (7,-1)
to the arrays (Node,Next) and modify the -1 entry in Next[4] to
point to the new position, 17.

```
        Offset            Node  Next
   1    1   -------------->  2    2
   2    3   ------------      3  -1
   3    5   ---------- |->  1    4
   4    9   -------- |       3   17 <--- Points to new nbr of 2
   5    10   ------ ||---> 1    6
   6    12   ---- ||      2   7
   7    14   -- |||       4   8
              ||||       6   9
              ||| ----->  3   -1
              || ------->  6   11
              ||          7  -1
              | --------->  5   13
              |            7  -1
               ----------->  5   15
                           6  -1
                           7  -1
```

You can also use more complicated data structures to represent
adjacency lists, if you are comfortable with them:

1. An array of pointers, each pointing into a linked list. Entry
   i in the array points to the list of neighbours of i.

2. Use C++ vectors.

===
=======
Shortest paths, revisited

Here is another way to compute shortest paths. We compute

   shortestpath(s,t,i) : shortest path from s to t that uses

at most i edges

Initially

shortestpath(s,t,0) = 0,   if s = t
                infinity, otherwise

                  / shortestpath (s,t,i),
shortestpath(s,t,i+1) = min (
                  \ min shortestpath(s,k,i) + W[k,t]
                    k

Note that the shortest path from s to t will never go through a loop, because we can remove the loop and get a shorter path. Thus, we only need to find paths of length at most N-1.

This is called the Bellman-Ford algorithm.

We can compute this by storing the values in a 3 dimensional array SP. Assume the vertices are numbered [1,2,...,n]

We maintain the value shortestpath(s,t,i) as SP[s,t,i].

Initially

   SP[s,t,0] = 0 if s = t, infinity otherwise

   for (i = 1; i < n; i++){
      for (t = 1; t <= n; t++){
        /*
          SP[s,t,i] = min (SP[s,t,i-1],
                      min SP[s,u,i]+W[u,t])
                       u
        */
        SP[s,t,i] = SP[s,t,i-1];
        for (u = 1; u <= n; u++){
          SP[s,t,i] = min(SP[s,t,i],SP[s,u,i-i]+W[u,t]);
        }
      }
   }

This algorithm takes time n^3 to compute SP[s,t,n-1]. Note that to compute SP[s,t,i], we only need SP[s,t,i-1], so we can make do

with just two 2-dimensional arrays of size n x n that we use
alternately, rather than a 3-dimensional array of size n x n x n.

Claim: Bellman-Ford can compute shortest paths in situations that
Dijkstra does not.

Suppose we can have negative edge weights. If we have cycles
with a total negative weight, the notion of shortest path does
not make sense because we can go around the negative cycle as
many times as we want and reduce the overall weight. So, let us
permit negative edge weights, provided they do not induce
negative cycles.

Will Dijkstra's algorithm work in graphs with negative edges?

Will How Bellman-Ford work if there are negative edges? How many
iterations will it take? (Note that since there are no negative
weight cycles, any cycle we visit will have a positive weight and
can hence be eliminated from any shortest path.)

==
=======

Wires and switches (IOI 95)

N points, M switches, each point is connected by a wire to one
switch, but two points may be connected to same switch.

```
          Points  Switches
            .      ./ .--
            .\    ./ .--|
   --Probe-> ._\__   ./ .--|---
   |        . \\__./ .--|  |
   |        .___\/ ./ .--|  |
   |        .  \__./ .--   |
   |                 |
    -----------------------------
```

All switches are simultaneously connected to a probe. We can
connect a probe to a point to check if circuit is complete with
current values of switches.

Two commands available:

    Flip(j)   --- Flip switch j
    Probe(i) --- Check if the circuit to point i is current complete

Goal:

For all points, determine the switch to which that point is
connected, using no more than 900 of Change and Probe commands.
Range for N is 1..90. Initially, all switches are off (open).

Brute force algorithm:

Probe 1. Close switch 1..M and stop when the probe succeeds.
Probe 2. Close switch 1..M and stop when the probe succeeds.
..   ..
..   ..
Probe N. Close switch 1..M and stop when the probe succeeds.

O(M*N) Changes and O(M*N) Probes --- in the worst case, all wires
are connected to switch M.

Better alternative:

Use binary search to find the switch for a given point --- that
is, switch on half the switches and probe the point:

  if on, then turn off half the switches
      else turn on half the remaining (need not turn off the first
      half!)

   This needs M Changes and log M probes

Overall N*M changes, and N*log M probes.

Can we now overlap the searches?

Switch on first half. Probe all points. This splits the points
into those that fall into the first half and those that fall into
second half.

Searches in two halves do not interact: Suppose initially we
switch on the first half of switches. After fully exploring all

points that lie in first half, we do not have to reset these
switches when we continue our binary search into the second half
because, for points connected to the second half, the switches of
the first half have no effect.

This solution can be analyzed to give a very simple algorithm.

Suppose we have 8 switches. We enumerate the binary numbers from
000 to 111 in order and arrange them in a column.

```
                 0 0 0
                 0 0 1
                 0 1 0
                 0 1 1
                 1 0 0
                 1 0 1
                 1 1 0
                 1 1 1
```

We set the switches according to the columns above and probe each
point once for each column setting. Suppose a point is connected
to switch 4. Then, the 3 experiments will yield answers "0 1 1".
Thus, for each point, the result of the experiments gives the
binary encoding of the switch to which it is connected.

Analysis:

M/2 log M Changes : there are log M columns and going from
                one column to another requires flipping
                half the switches

N log M Probes : probe N points for each column

==
=======

Triangulation

We have a convex polygon (a polygon in which any line joining a
pair of interior points stays inside the polygon).

Here are examples of convex and non convex polygons

```
    1          6---1          1----------2
   / \        / |             |         |
  5    2     5  |             |  4-----3
  |    |     |  2             |  |
  4 ----- 3  | /              |  |
             4---3            6----5
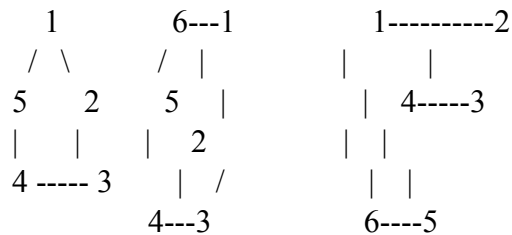
      Convex                  Nonconvex
```

A triangulation consists of partition the polygon into triangles
whose vertices are the vertices of the polygon such that no
triangles overlap.

Here are some triangulations of a hexagon.

```
   -- 1 --         -- 1 --           -- 1 --
  /      \        /      \          / /\ \
 6 -------- 2    6 -------- 2      6 / \  2
 |    ___//|     |_____/|       | / \ |
 | _/  / | |     |/        |       |/    \|
 5/   / 3       5 ---------- 3     5 ---------- 3
  \  / /         \        /         \        /
   -- 4 --         -- 4 --           -- 4 --
```

Notice that we only require the vertices of each triangle to
coincide with vertices of the orignal polygon. The third
triangulattion shows a triangle (1,3,5) none of whose edges
coincide with the polygon

Counting the number of triangulations of an n vertex polygon:

Let f(n) be the number of triangulations of a convex polygon with
n vertices. How can we write a recurrence for f(n)?

If we fix an edge (n,1), it forms a triangle using one other
vertex i.   For this fixed triangle, we have two sub polygons,
one with vertices 1...i and the other with vertices i...n.

```
           ......
          /      \
```

```
     2        i
     |    ____// \
     | __/   /
     1 /   /   .
       \   /   .
        -- n --...
```

These can be triangulated in f(i) and f(n-i) ways so this
particular choice of triangle generates f(i)*f(n-i)
triangulations. We can vary i from 2 to n-1 to get the
recurrence

```
  f(n) =   sum      f(i) * f(n-i)
        i = 2 to n-1
```


===
=======

Optimal Triangulation

Given the coordinates of the vertices of a convex polygon,
compute a triangulation that minimizes the total length of edges
used in the triangulation.

Solution:

Assume vertices are numbered clockwise 1..n.

Best(i,j) is the best way of triangulating the polygon formed by
vertices i,i+1,...,j and diagonal i--j. If we triangulate this
polygon, the diagonal i--j is connected to some vertex k between
i and j.

```
      *......i...
      /    /  .
     *    /    .
     |   /   k
     |  /  __/ .
     * / __/   .
      \/__/    .
       j-- .......
```

Now we have the recurrence:

$$\text{Best}(i,j) = \min_{i < k < j} \text{Best}(i,k) + \text{Best}(k,j) + d(i,k) + d(k,j)$$

Base case:

$\text{Best}(i,j) = 0$ if $j = i+2$

The answer we want is $\text{Best}(1,n)$

===

Heaps

Dijkstra's algorithm requires us, in each round, to

1. Pick a vertex with minimum burning time
2. Change the expected burning time for each neighbour

If we use an adjacency list representation, step 2 will be efficient (note that in this adjacency list we need to store the edge weight as well).

How do we do step 1 efficiently?

Keep the list sorted:

-- If the list is an array

  Search is fast (log n)
  Insert is slow (n)

-- If the list is a linked list

  Search is slow (n)
  Insert is fast (constant time) <-- after we know where to insert

Instead, use a different data structure.

A heap is a binary tree with two properties:

1. The tree is filled level by level, without any gaps, so it is
   a complete binary tree where only the bottom most level may be
   incomplete.

2. The value at each node is less than or equal to that of its
   parents.

Here is an example of a heap.

```
        ---3---
        /     \
      14       11
     /  \     /  \
   16   19  13    15
  / \  / \
 21 20 23 24
```

Suppose we want to insert an element, say 10, into the heap.

Since we must fill the heap level by level, we can only insert it
in the first free position on the bottom row.

```
        ---3---
        /     \
      14       11
     /  \     /  \
   16   19  13    15
  / \  / \   /
 21  20 23  24 10
```

This violates the heap property, but only between 10 and its
parent.   We can fix the problem by swapping 10 with its parent.

```
        ---3---
        /     \
      14       11
     /  \     /  \
   16   19  10    15
  / \  / \   /
 21  20 23  24 13
```

This transfers the problem one level up, so we swap again and then stop because there are no further heap violations.

```
        ---3---
        /     \
      14       10
     / \      / \
   16  19   11   15
  / \ / \   /
 21 20 23 24 13
```

The height of such a complete binary tree with n nodes is at most log n, so we have to "bubble up" the new value at most log n times till the heap is restored to a stable state.

Clearly, the minimum value in a heap is found at the root. Thus, assuming we can compute the parent and children of a node efficiently, we can:

  insert a vertex in log n time
  find the minimum in constant time

But, in Dijkstra's algorithm, we also want to delete the minimum vertex so that it is not considered again later on.

If we delete the minimumv value 3 from our heap, we create a hole at the root. How do we restore the structure?

```
        --[ ]--
        /     \
      14       10
     / \      / \
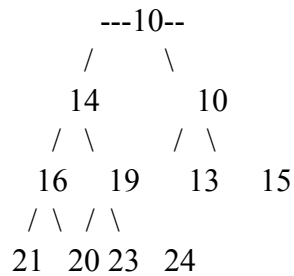   16  19   11   15
  / \ / \   /
 21 20 23 24 13
```

In the new heap, we have one less element, so eventually the last slot currently used in the bottom row will be vacant. So, one option is to just move this last value to the root.

```
        ---13--
        /      \
      14         10
     / \        / \
   16   19   11    15
  / \ / \
 21  20 23  24
```

Now, we have a possible violation of heap property at the root. We examine 13 and its two children and swap it with the smaller of its children, to get the following tree.

```
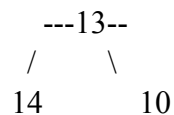        ---10--
        /      \
      14         13
     / \        / \
   16   19   11    15
  / \ / \
 21  20 23  24
```

This pushes the heap violation one step down. Again we swap 13 with the smaller of its children, to get

```
        ---10--
        /      \
      14         10
     / \        / \
   16   19   13    15
  / \ / \
 21  20 23  24
```

Now we stop. Once again, the number of steps we take is proportional to the height of the tree, so delete-min takes time log n.

What about updating a value?

Suppose we change 19 to 6?

```
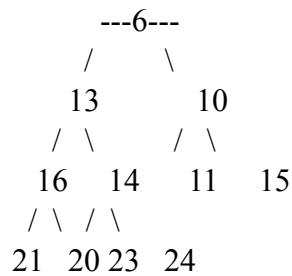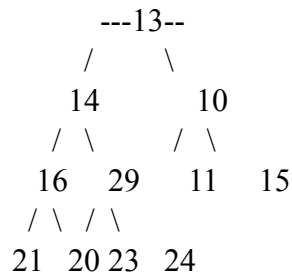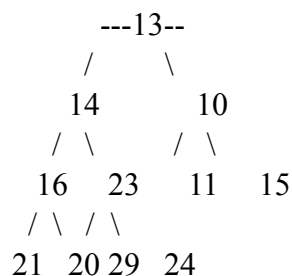        ---13--
        /      \
      14         10
```

```
         /  \        /  \
       16    6     11     15
      /  \  /  \
    21   20 23  24
```

The heap property is violated above 6, so we move it up as we do
in insert, to get

```
             ---6---
            /       \
          13          10
         /  \        /  \
       16    14    11     15
      /  \  /  \
    21   20 23  24
```

Suppose, instead, we increased the value at 19 to 29, instead of
reducing it to 6?

```
             ---13--
            /       \
          14          10
         /  \        /  \
       16    29    11     15
      /  \  /  \
    21   20 23  24
```

In this case, we push it down, as in delete-min, till we get a
heap.

```
             ---13--
            /       \
          14          10
         /  \        /  \
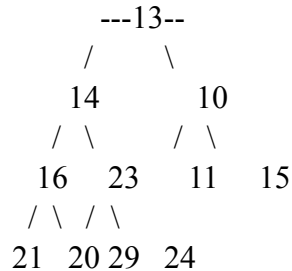       16    23    11     15
      /  \  /  \
    21   20 29  24
```

So, changing a value and restoring the heap structure can also be
done in log n time.

To summarize, assuming we can find the parent/child of a node
efficiently and also find the last element of a heap efficiently,

we can perform the following operations efficiently:

```
 find-min    : constant time
 insert      : log n time
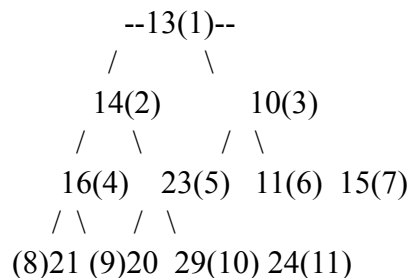 delete-min  : log n time
 update-value : log n time
```

How do we store a heap so that we can manipulate it easily?

```
            ---13--
           /       \
         14          10
         / \        / \
       16    23   11    15
      / \  / \
    21  20 29  24
```

We read off the values, level by level, and store them in an array. The heap above would be an array of the form

```
 [ 13  14  10  16  23  11  15  21  21  20  29  24]
```

Here is a picture of the heap where each value is labelled by its position in the array.

```
            --13(1)--
           /         \
        14(2)         10(3)
        /   \         / \
     16(4)   23(5)  11(6)  15(7)
     / \     / \
  (8)21 (9)20  29(10) 24(11)
```

From this, it is immediate that the children of a node at position j are found at positions 2j and 2j+1. Likewise, the parent of a node a position j is at position floor(j/2), where floor(n) is the largest integer less than or equal to n.

One way to construct a heap is to insert the elements one at a time. This takes n log n time, since we insert n elements and each insert takes log n time.

Exercise: Given an array, make into a heap in place (without

using another array) in time n.

Using a heap to sort

Once we have a heap, we know that the minimum element is at the
root. If we print it out and delete it, the second smallest
element rises to the root. Thus, by repeatedly deleting the
minimum element, we can generate the values in ascending order.

It takes time n log n to create a heap and n log n to do n
delete-min operations, so this gives us an n log n algorithm for
sorting.

Max-heaps and min-heaps

The heaps we have considered are called min-heaps, because the
minimum value is at the root. If we reverse the second heap
condition---that is, we write

2'. The value at each node is greater than or equal to that of
    its parents.

we will get the largest value at the top of the heap. This is
called a max-heap. We can define find-max, insert, delete-max
and update for max-heaps along the same lines as we did for
min-heaps.

===
=======
===
=======
        IOI Training Camp, Lecture Notes, Day 3

                Sun 22 Jun 2008
===
=======
Session 1, 09:00--10:30
===
=======

Heaps

Exercise: Given an array, make into a heap in place (without using another array) in time n.

Solution:

Write out the contents of the array in the form of a binary tree.

For example, if we start with 7 5 6 15 2 1 18, the tree is

```
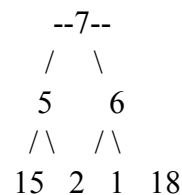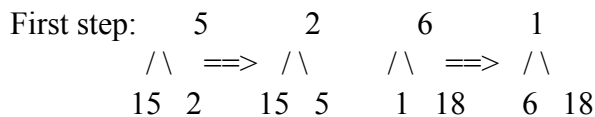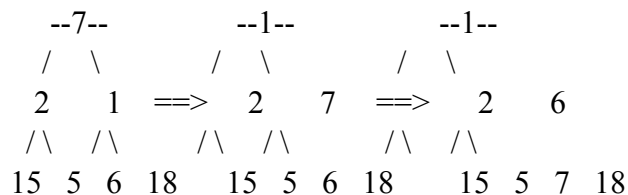        --7--
        /   \
       5     6
      /\    /\
    15  2  1  18
```

Notice that the nodes at the bottom level satisfy the heap property, since they don't have any children. Starting at these "leaf" nodes, we work up the tree, combining heaps at the next level.

Each lowest level node (index greater than n div 2) is a singleton heap. For each node at level at second lowest level, we locally adjust the values into 3 element heaps. At the next level, we combine two 3 element heaps with a new root into 7 element heaps. Each higher level has 1/2 the number of nodes.

```
First step:    5         2        6         1
              /\   ==>  /\        /\   ==>  /\
            15  2      15  5     1  18     6  18
```

Second step:

```
      --7--          --1--          --1--
      /   \          /   \          /   \
     2     1   ==>  2     7   ==>  2     6
    /\    /\       /\    /\       /\    /\
  15  5  6  18   15  5  6  18   15  5  7  18
```

What is the complexity?

Naively, it looks like n log n since we fix the heap n/2 times and each fix takes log n.

To fix the heap at the root of n nodes, we need to make a heap at each of its children (trees of size n/2) and then fix the root (walk down upto log n steps flipping values).

Notice that later steps take more operations but there are only half as many corrections to make. Overall, if you add up the operations, it turns out to be O(n).

Dijkstra's algorithm with a heap

ExpectedBurnTime[1] = 0;
ExpectedBurnTime[i] = infinity, for i other than 1
                (Use a sufficiently large value, say the
                 sum of all the weights in the graph)

Burnt[i]   = 0, for all i

In each round,

Among all unburnt vertices, pick j with minimum
ExpectedBurnTime. (Initially, this is 1 since Burnt[i] = 0
for all i and only ExpectedBurnTime[1] is not infinity

Set Burnt[j] = 1.

Update ExpectedBurnTime[k] for each neighbour k of j, if
necessary.

The number of rounds is N, the number of vertices, since we pick
a vertex and process it in each round.

If we use adjacency matrix, each round takes time N because

(a) it takes time N to find the minimum unburnt value
(b) it takes time N to scan all neighbours

We can fix the complexity of (b) by using an adjacency list
instead of an adjacency matrix.

To fix (a) we keep the values of the form (v,ExpectedBurnTime) of
unburnt vertices in a heap. This allows us to find the minimum
unburnt vertex in log n time.

Now, we still have to update the ExpectedBurnTime of all its
neighbours. Suppose w is an unburnt neighbour of v. We know we can update the
ExpectedBurnTimed of w in log n time, but how do we find w in the
heap? If we scan the array searching for a value (w,i), it will
again take time N to update values.

To get around this, we maintain extra information telling us
where each vertex sits in the heap. Maintain an array Index such
that Index[v] = i means that the value for vertex v is at
position i in the heap. Note that we have to update entries in
Index each time we swap values while inserting, deleting or
modifying values in the heap.

What is the complexity?

Naively, a vertex could have N-1 neighbours, so each round can
take N log N time.

However, let us count more carefully. How many times will the
value of vertex w be updated? Once for each of its neighbours.
In other words, the value of w will be updated degree(w) times.
If we sum up the degrees of all the nodes in the graph, we get
2*M where M is the number of edges in the graph (each edge is
counted twice, once at each end point).

So, overall, we make M updates, each taking log N time, so this
version takes 2 M log N + N log N = M log N

M could be N^2 in which case M log N is N^2 log N, which is worse
than the complexity of the original implementation, so we should
use the heap version only when M is small compared to N^2 log N

===
=======
Speculating on buffalos

Ramu was a lazy farmer. He had inherited a fairly large farm and
a nice house from his father. Ramu leased out the farm land to
others and earned a rather handsome income. His father used to
keep a buffalo at home and sell its milk but the buffalo died a
few days after his father did.

Ramu too wanted to make some money from buffaloes, but in a quite
a different way. He decided that his future lay in speculating on
buffaloes. In the market in his village, buffaloes were bought
and sold everyday. The price fluctuated over the year, but on any
single day the price was always the same.

He decided that he would buy buffaloes when the price was low and
sell them when the price was high and, in the process,
accummulate great wealth. Unfortunately his house had space for
just one buffalo and so he could own at most one buffalo at any
time.

Before he entered the buffalo market, he decided to examine to
examine the variation in the price of buffaloes over the last few
days and determine the maximum profit he could have made. Suppose
the price of a buffalo over a period of 10 days varied as

```
-------------------------------------------------
 Day  : 1  2  3  4  5  6  7  8  9  10
-------------------------------------------------
 Price : 11  7  10  9  13  14  10  15  12  10
-------------------------------------------------
```

The maximum money he can make in this case is 13 ---
 buy on day 2, sell on day 3 (+3)
 buy on day 4, sell on day 6 (+5)
 buy on day 7, sell on day 8 (+5)

How do we arrive at an optimal strategy?

Plot a graph of buffalo prices as they vary day by day.

```
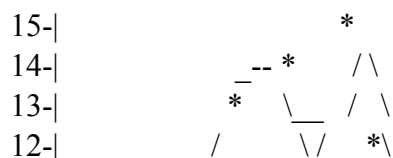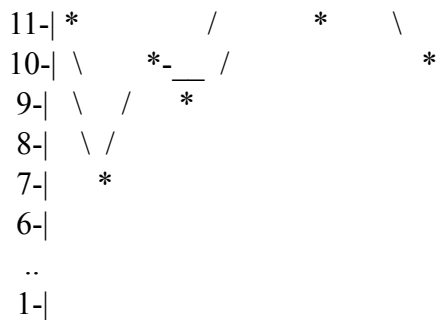 15-|                        *
 14-|              _-- *      / \
 13-|          *   \__   /  \
 12-|         /      \/   *\
```

```
11-| *            /        *      \
10-| \       *-__ /               *
 9-| \   /      *
 8-|  \ /
 7-|    *
 6-|
  ..
 1-|
  ----------------------------------------------------
  Day   1  2  3  4  5  6  7  8  9  10
```

Profit making intervals are where the graph goes up.

  In this case, profit making intervals are days 2-3, 4-6, 7-8

Buy whenever the
price starts increasing and sell whenever the price starts
decreasing. Exploit all these intervals to maximize profit.

==
=======
Speculating on buffalos, Version 2

To discourage speculation, the panchayat decides to charge a
transaction fee. Each time a buffalo is bought or sold, Rs 2 has
to be paid to the panchayat. What should Ramu's strategy be now?

```
--------------------------------------------------
 Day  : 1  2  3  4  5  6  7  8  9  10
--------------------------------------------------
 Price : 11  7  10  9  13  14  10  15  12  10
--------------------------------------------------
```

Example:

  Buy at 2, sell at 6 --- profit is (14-7) - 4 = 3
  But at 7, sell at 8 --- profit is (15-10) - 4 = 1

Can try a number of ad hoc strategies, but it is difficult to
argue that such a strategy is correct. Instead, adopt a
systematic way to calculate the best profit, like we did for grid
paths.

Let Profit[i..N] be the money Ramu can make on days i..N,
assuming he has no buffaloes at the start of day i.

On day i, Ramu can either buy or not buy a buffalow.

If Ramu does not buy on day i, it is as though Ramu started his
transactions directly on day i+1. So,

   Profit[i..N] = Profit[i+1..N]

If Ramu does buy on day i, the money he makes depends on when he
sells the buffalo:

  If he sells on day i+1, the money he makes is

    Price(day i+1) - Price(day i) - 4 + Profit[i+2..N]

  If he sells on day i+2, the money he makes is

    Price(day i+2) - Price(day i) - 4 + Profit[i+3..N]

  ...

  If he sells on day N-2, the money he makes is

    Price(day N-2) - Price(day i) - 4 + Profit[N-1..N]

  If he sells on day N-1, the money he makes is

    Price(day N-1) - Price(day i) - 4 + Profit[N]

    (Notice that Profit[N] is zero since he cannot buy and sell
     a buffalo in one day).

  If he sells on day N, the money he makes is

    Price(day N) - Price(day i) - 4

Ramu's best strategy is to calculate the maximum of all these
quantities.

  Profit[i..N] = max (Profit[i+1..N],

Price(i+1) - Price(i) - 4 + Profit[i+2..N],
                        Price(i+2) - Price(i) - 4 + Profit[i+3..N],
                        ...
                        Price(N-2) - Price(i) - 4 + Profit[N-1..N],
                        Price(N-1) - Price(i) - 4 + Profit[N],
                        Price(N) - Price(i) - 4
)

Using this we can compute explicitly

  Profit[N-1] = max (Profit[N],
            Price(N) - Price(N-1) - 4)

Once we know Profit[N] and Profit[N-1], we can compute Profit[N-2]
explicitly

  Profit[N-2] = max (Profit[N-1],
            Price(N-1) - Price(N-2) - 4 + Profit[N],
            Price(N) - Price(N-2) - 4)

Continuing in this vein, we can compute Profit[N-3], Profit[N-4],
..., Profit[i].

In general, the final answer that we want is Profit[1].

A linear time algorithm:

  With[i]   --- max profit starting i, if Ramu already has a buffalo
  Without[i] --- max profit starting i, if Ramu does not have a buffalo

  With[i]   = max { With[i-1], Without[i-1] - Price(i) - 2 }
    // If Ramu has a buffalo at i, either he already had it at
    // i-1 (With[i-1]), or he did not have it at i-1 and bought
    // it at i (Without[i-1] - Price(i) - 2)

  Without[i] = max { Without[i-1], With[i-1] + Price(i) - 2 }

===
=======

Maximum sum subsection

Given a sequence of integers, a subsection is a block of
contiguous integers in the sequence. The task is to find the
subsection whose sum is maximum.

For instance, here is a sequence and a subsection marked out
whose sum is -1

```
 3 -1  2 -1  2 -3  4 -2  1  2  1
   |<------------>|
      sum = -1
```

Exercise: Find an O(N) algorithm for this problem

```
  Best(i)  = Maximum sum segment ending at i

  Best(i) = max { A[i],             // Best segment starts here
            A[i] + Best(i-1)    // Best segment extends previous
          }
  Best(1) = A[1]
```

This assumes subsections are of length > 0. If zero length
segments are permitted, we also have to add a term 0 to the max
term when computing Best(i). (For instance, if all values are
negative, the maximum sum subsection would then be the empty
subsection with sum 0.)

==
=======

Post Office (IOI 2000)

There is a straight highway with villages alongside the
highway. The highway is represented as an integer axis, and the
position of each village is identified with a single integer
coordinate. There are no two villages in the same position. The
distance between two positions is the absolute value of the
difference of their integer coordinates.

Post offices will be built in some, but not necessarily all of
the villages. A village and the post office in it have the same

position. For building the post offices, their positions should be chosen so that the average distance from each village to its nearest post office is minimized.

You are to write a program which, given the positions of the villages and the number of post offices, computes the least possible sum of all distances between each village and its nearest post office, and the respective desired positions of the post offices.

Solution

Minimizing the average distance is the same as minimizing the sum of the distances from each village to its nearest post office.

N villages at positions $1 \le v_1 \le v_2 \le ... \le v_N$

Want to place K post offices at K of the N positions $v_1..v_N$.

Suppose we try to go from left to right, placing post offices and maintain Best[i,r], the best value of placing r post offices among $v_1,...,v_i$.

But, if we add a new post office at v_{i+1}, the distances for the first i villages may change because the nearest post office for the last few villages in $v_1,...,v_i$ may now be now be v_{i+1} rather than the rightmost post office in $v_1,...v_i$.

Hint: Try to strengthen the condition for Best[i,r].

  Best(i,r) : best way of placing r post offices among $v_1,...v_i$
        such that there is a post office at v_i.

How do we calculate Best(i,r)?

  Best(i,r) = min{ Best(i-1,r-1)   // prev PO at i-1
            Best(i-2,r-1) + Cost(i-1) // Prev PO at i-2
            Best(i-3,r-1) + Cost(i-2,i-1) // Prev PO at i-3
              ...
          }

Thus,

Best(i,r) = min   Best(j,r-1) + Cost(j+1,..,i-1)
        1 <= j < i-1

Here Cost(j+1,...,i-1) gives costs for villages v{j+1}...v{i-1}
given that nearest neighbouring post offices are at vj and vi.

What is the final answer that we want?

We have to decide where we place the last post office and compute
cost for villages to the right of that. Thus, the final answer
is given by the expression:

    min   { Best(j,K) +   Sum   distance(vj,vl) }
1 <= j <= N   j+1 < l <= N

Complexity?

N^2 K

===
==
Phidias (IOI 2004)

Phidias is a sculptor who has a large rectangular slab of marble
of size W x L (1 <= W,L <= 600). He wants to cut out rectangular
tiles of specific sizes (a1,b1), (a2,b2) ... (ak,bk). He is
willing to take as many of each type as are available. The
marble slab has a pattern and so tiles cannot be rotated. This
means that it is not acceptable to produce a tile of (bi,ai)
instead of (ai,bi).

To cut a slab of marble, he can can make a full horizontal or
vertical cut across the slab at an integer point.

After making a sequence of cuts, Phidias will arrive at some
pieces of marble that are wasted --- they cannot be cut down to
any of the desired tiles.

Find a sequence of cuts that will minimize the wastage.

Solution:

Wastage(x,y) is min wastage for x by y marble piece

```
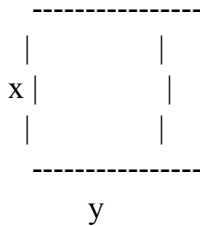        ----------------
        |       |
      x |       |
        |       |
        ----------------
              y
```

If we cut vertically at yi, we get

  Wastage(x,y) = Wastage(x,yi) + Wastage(x,y-yi).

Similarly, if we cut horizontally at xj, we get

  Wastage(x,y) = Wastage(xj,y) + Wastage(x-xj,y).

The choices available to us are to cut vertically at yi for
$1 <= yi < y$ or horizontally at xj for $1 <= xj < x$.

We check all of these and choose the minimum.

Initially:

Wastage(x,y) = 0,   if (x,y) matches some (ai,bi)
          = x*y, otherwise (this will be updated later)

Update:

Wastage(x,y) =

min { Wastage(x,y),   /* Currently computed quantity */

      min   Wastage(j,y) + Wastage(x-j,y),
    $0 < j < x$ /* Horizontal cut at j */

      min   Wastage(i,y) + Wastage(x-i,y),
    $0 < i < y$ /* Vertical cut at i */
    }

Alternatives:

Initialize only

  Wastage(x,y) = 0,   if (x,y) matches some (ai,bi)
  Wastage(1,1) = 1

If (x,y) is completely wasted, all cuts will eventually create a
collection of (1,1) rectangles which add up to x*y.

Can optimize a bit by adding

  Wastage[x,y] = x*y whenever no tile fits,
             that is (x < min xi) or (y < min yj)

===
=======
Mexico (IOI 2006)

Mexico City is built in a beautiful valley known as the Valley of
Mexico which, years ago, was mostly a lake. Around the year 1300,
Aztec religious leaders decreed that the lake's center be filled
in order to build the capital of their empire. Today, the lake is
completely covered.

Before the Aztecs arrived, c cities were located around the lake
on its shores. Some of these cities established commercial
agreements. Goods were traded, using boats, between cities that
had a commercial agreement. It was possible to connect any two
cities by a line segment through the lake.

Eventually, the kings of the cities decided to organize this
commerce. They designed a commerce route that connected every
city around the lake. The route met the following requirements:

- It could start in any of the cities, visited each of the cities
  around the lake, and finally ended in another city different
  from the starting city.

- The route visited each city exactly once.

- Every pair of consecutively visited cities in the route had a
  commercial agreement.

- Every pair of consecutively visited cities in the route was
  connected by a line segment.

- To avoid crashes between boats, the route never crossed itself.

The figure below shows the lake and the cities around it. The
lines (both * and -) represent commercial agreements between
cities. The lines with * represent a commerce route starting in
city 2 and ending in city 5.

This route never crosses itself. It would not be legal, for
example, to construct a route that went from 2 to 6 to 5 to
1, since the route would cross itself.

```
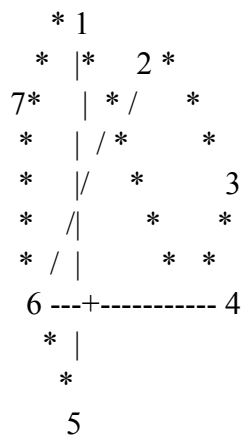       * 1
      *  |*    2 *
    7*   | * /    *
     *   | / *       *
     *   |/   *        3
     *   /|     *      *
     * / |         *  *
      6 ---+----------- 4
        * |
          *
          5
```

Cities in the lake are numbered from 1 through c moving in
clockwise direction.

Write a program that, given both the count c of cities and a list
of the commercial agreements between them, constructs a commerce
route that meets the above requirements.

Solution:

Pick a pair of cities (u,v) around the lake such that there is an
edge (u,v). Assuming this is part of the final route, this

divides the circle into two disjoint parts. Solve them
separately. In one part, find a solution that ends at u and in
the other part, find a solution that ends at v. The
sub-solutions will not cross each other or cross the edge (u,v),
so we can combine these sub-solutions with the edge (u,v) to get
a solution overall.

This solution will work, in general, but it becomes difficult to
program because at recursive steps we have to maintain the
current list of cities along the periphery that are being handled
in the recursive call.

Instead, we adopt a different approach (which is NOT the same as
the earlier divide-and-conquer).

Let i and j be any distinct cities along the periphery and
consider the ordered pair (i,j). We define

  Clockwise(i,j) : there is a valid tour among the cities in the
          arc i,i+1,...,j ending at j

  Anticlockwise(i,j) : there is a valid tour among the cities in
          the arc j,j+1,...,i ending at j

Let us write an inductive definition for Clockwise(i,j). Since
the path ends at j, the last edge in the path is either (j-1,j)
or (i,j). Any other edge to j would cut through the centre of
the lake and not allow a nonintersecting path among all cities.

If the last edge in the path Clockwise(i,j) is (j-1,j), there is
an edge (j-1,j) and, inductively, a path in i,i+1,...,j-1 ending
at j-1. This is the same as Clockwise(i,j-1).

On the other hand, if the last path in Clockwise(i,j) is (i,j),
we need a path in (i,...,j-1) ending at i to which we add (i,j)
to get the overall path. This requirement can be expressed as
Anticlockwise(j-1,i).

Thus, we have

  Clockwise(i,j) = Clockwise(i,j-1) AND Edge(j-1,j)
          OR
          Anticlockwise(j-1,i) AND Edge(i,j)

Symmetrically, for Anticlockwise(i,j), we either end with an edge (j+1,j) or an edge (i,j). Similar to the above, we have

  Anticlockwise(i,j) = Anticlockwise(i,j+1) AND Edge(j+1,j)
          OR
                  Clockwise(j+1,i) AND Edge(i,j)

Notice that whenever we refer to Clockwise(i,j) or Anticlockwise(i,j) in this formulation, we are always referring to the entire segment between i and j (or j and i) along the circumference, so there is no problem keeping track of which cities are part of the subproblem being considered.

The base cases are

  Clockwise(j,j+1) = True iff Edge(j,j+1)
  Anticlockwise(j+1,j) = True iff Edge(j,j+1)

Finally, we have a solution to the problem if there exist a city i such that

  Clockwise(i+1,i) (or equivalently, Anticlockwise(i+1,i))

Here, i would be one end point of the valid route.

Alternatively, we could look for i and just such that

  Clockwise(i,j) AND Anticlockwise(i-1,j)

Both these paths end in j and can be combined into a nonoverlapping path for all the cities.

==
=======
Session 2, 11:00--12:30
==
=======

min-Segments

You are given a sequence of N integers x1, x2, ..., xN, where N

can be as large as 10^6. You are also given an interval length
M, where M can be as large as 10^5. The goal is to report the
minimum of every segment of M numbers in the list.

In other words, the output should be

   min (x1,x2,...,xM)
   min (x2,...,xM,xM+1)
   min (x3,...,xM+1,xM+2)
   ..
   min (x(N-M+1),...,xN-1,xN)

If we compute the minimum explicitly for each M segment of the
input, we end up doing about O(NM) work, which is too much given
the limits for each N and M.

Can we do better?

Solution:

Break up the sequence into segments of length M.

For each segment

     y1 y2 ... yi ... yM

compute and store for each yi the two quantities
min(y1,y2,...,yi) and min(yi,yi+1,...,yM). This can be done in   good idea.
two linear scans of y1,y2,...,yM, one from left to right and one
from right to left.

Now, if we ask for the minimum of some arbitrary interval of
length M, it will in general span across two of the fixed
intervals for which we have precomputed the values above.

     y1 y2 ... yi-1 yi ... yM z1 z2 ... zi-1 zi ... zM
                   | |
            <---------------------->

We can compute min(yi,...,yM,z1,...,zi-1) from the precomputed
values for these two intervals as

   min(min(yi...,yM), min(z1,...,zi-1))

Thus, in one scan, we can use the precomputed values to compute the minimum for all values.

How much time does the precomputation take? We have to precompute values for N/M intervals. Each interval requires O(M) work, so overall we take about O(N) time for precomputation.

The final solution then takes another O(N) scan of the sequence with the precomputed values.

Notice that this technique can be applied to any function that is associative --- e.g. max, gcd ...

It can also be applied in two dimensions. Suppose we have a large N x N array and we are looking for the maximum value in every M x M subarray of the large array.

We tile the N x N array into M x M arrays and for each (i,j) we compute the maximum with respect to the four small rectangles that it forms inside its M x M block.

```
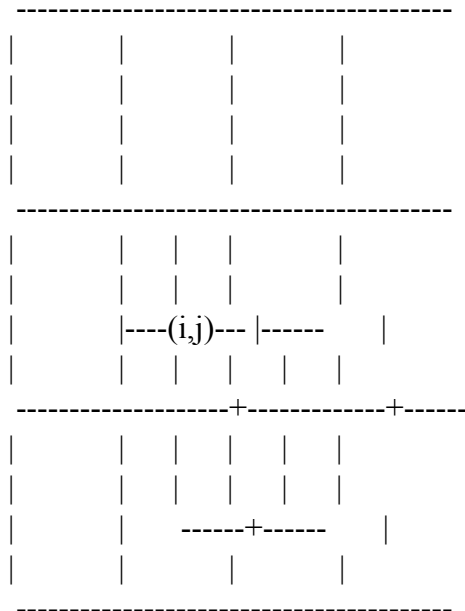     ------------------------------------------
     |         |         |         |          |
     |         |         |         |          |
     |         |         |         |          |
     |         |         |         |          |
     ------------------------------------------
     |         |   |   |          |
     |         |   |   |          |
     |         |----(i,j)--- |------       |
     |         |   |   |   |   |
     -------------------+------------+------
     |         |   |   |   |   |
     |         |   |   |   |   |
     |         |      ------+------       |
     |         |         |          |
     ------------------------------------------
```

Now, for an arbitrary M x M block that spans four of the original tiles, we combine the precomputed values for the four smaller rectangles that make up this M x M block.

===
=======

Race (CEOI 2003) ---------------

250,000 rockets racing in parallel tracks across space. Each
rocket i has a starting position x_i and a constant velocity v_i
(bounded by 100) with which it will move. Assume that the
rockets are enumerated in order of their starting position.

Thus, initially the rockets are lined up as:

-----x-->
   (x1,v1)

---------x-->
      (x2,v2)

----------------x--->
          (x3,v3)

....

--------------------------x----------------------------------->
              (x250000,v250000)

Constraints:

At any given time, no more two rockets are at the same position.
This implies that all overtakings take place at distinct times.

The race goes on forever. Eventually, faster rockets overtake
slower rockets and beyond a point the rockets will become ordered
in the order of their velocities.

Problems
----

1. Compute the number of overtakings (modulo 1000000, say) in the
   eventual race.

2. Print out the first 10,000 overtakings (in order).

Solution

Part 1

For starting position x_i, the rocket at that position will eventually overtake every rocket starting at x_j > x_i with velocity v_j < v_i. For each each velocity v_j maintain an array A_j of size 250000 which records for each position j the number of rockets of velocity v_i from position j to 250000 in the initial order. Then, to compute the number of rockets overtaken by the rocket (x_i,v_i), just add up A_j[i] for each v_j < v_i.

Naively, this takes time N^2. We can do it more efficiently as follows.

Maintain an array AtV[1..100] that records for each velocity v, the number of rockets AtV[v] that we have seen at velocity v.

Initialize AtV[1..100] to 0. Start from the last rocket. For rocket ri with position xi and velocity vi, increment AtV[vi]. Rocket ri will overtake all rockets to its right with smaller velocity. Inductively, AtV[] currently has the count all the velocities for all rockets to right of ri. So, add up AtV[1] + AtV[2] + ... + AtV[vi-1] to get number of overtakings that ri performs.

In this way, in one scan, we can add up the total number of overtakings. We scan N rockets and each step requires scanning the array AtV[] of size V, so this takes time N*V.

Another approach is to observe how the order of rockets changes. The final order is sorted in descending order terms of velocity. Each rocket r_i has a rank F_i in the final order and an initial rank S_i in the starting order.

How do we find the rank F_i for a rocket r_i? We have to sort the velocities, but this should be a stable sort so that rockets with same velocity retain their order. What we need to do is to count the number of swaps that between the original order and the stably sorted order.

We can use divide and conquer to get an O(N log N) solution to

the number of swaps. Break up the total list into two halves, recursively compute the number of overtakings in each half and then compute the number of overtakings from left half to right half.

(This is essentially merge sort---see separate notes on sorting.)

Part 2

From the initial configuration, we can compute for each rocket the identity of the next rocket it will overtake.

Observe that a rocket at (x_i, v_i) can overtake a rocket at (x_j, v_j) only if $x_j > x_i$ and $v_j < v_i$. In particular, among all rockets of this form, there is exactly one that it will overtake first. At any point, let order of the rockets be $1, 2, .., N$. Then the next overtaking must be an adjacent pair $(i, i+1)$. (Why?)

Consider all pairs (r_i, r_{i+1}) such that the next rocket that r_i can overtake is r_{i+1}. For each such pair, there is a time t_i at which the overtaking will occur. It is possible that r_i will never overtake r_{i+1} in which case we set t_i to infinity.

The next overtaking that will take place is the pair (r_i, r_{i+1}) with the shortest time t_i. We can thus maintain the pairs in a heap in terms of the time t_i at which the overtaking takes place.

At each step, we obtain the next overtaking by deleting the minimum element from the heap. This interchanges the pair $(i, i+1)$ in the current order of the rockets and renumbers $i+1$ as i' and i as $i'+1$. We then compute the times for $(i-1, i')$ and $(i', i+2)$ and add the two new elements to the heap.

What about the old entries for $(i-1, i)$ and $(i+1, i+2)$ which no longer reflect consecutive entries in the list? Should these entries be deleted? Deleting arbitrary elements from a heap is not easy. However, we do not really need to delete these entries. They are valid overtakings and will take place at the time appointed.

The problem that may occur is that we make duplicate entries.

For instance, suppose after the initial (i,i+1) overtaking, the
next overtaking is (i-1,i') (where i' is the old i+1). Now the
old (i-1,i) is again adjacent and we will introduce a duplicate
entry for this pair in the heap.

However, notice that we are promised that all overtaking times
are distinct. So, when we extract the minimum element from the
heap, if it is the same as the previous element we got, we can be
sure it is a duplicate and discard it.

Requires 250000 log 250000 operations to set up the heap and
10000*3*log 250000 operations to print out first 10000 entries.

==
=======

Sokoban (ACM)

A grid of squares with some blocked positions in which there is a
Sokoban player (S) and a block (B). The player S has to move the
block B to a target square T. Assume that the grid is at most 20
x 20.

Here is an example maze

```
        ###########        S -- You
        #T##......#      B -- Box
        #.#.#..####        # -- Blocked square
        #....B....#      T -- Target
        #.######..#        . -- Empty cell
        #.....S...#
        ###########
```

The only way to move the block is to push it. So, there are two
types of moves for S:

-- Walk east/west/north/south without pushing : Represent as e/w/n/s
-- Push east/west/north/south: Represent as E/W/N/S

To push, S must be oriented correctly with respect to B. Thus,
to push B west, S must be one square to the east of B. A push
moves both S and B one step in the direction of the push.

Find a way to move B to T that minimizes the number of pushes.
If two solutions have the same number of pushes, find the one
with the minimum number of walks.

For example, here is a solution for the grid above:

  eennwwWWWWeeeeeesswwwwwwwwnnNN

Solution:

This is clearly a shortest path problem in a graph. What is the
graph?

-- Vertices are (Sx,Sy,Bx,By)

-- For each outgoing vertex, there are four possible outgoing
   edges of which at most one is a push. The rest are walks.

   (Note that this is a directed graph --- if we go from
   (Sx,Sy,Bx,By) to (S'x,S'y,B'x,B'y), we may not be able to
   reverse this move, in particular if the move was a push.

   Note that all our earlier algorithms like BFS, DFS, shortest
   paths work for directed graphs as well, in essentially the
   same way as for undirected graphs.)

-- Target node is any vertex (Sx,Sy,Bx,By) where (Bx,By) =
   (Tx,Ty). If we assume that the initial box position is not
   the target, we have only four possibilities for (Sx,Sy).

-- Maintain the cost of a path as a pair (P,W) where P is the
   number of push move and W is the number walks. So, a push
   move adds cost (1,1) while a walk move adds (0,1). Costs are
   ordered lexicographically: (P,W) < (P',W') if P < P' or P = P'
   and W < W'.

   We can also map (P,W) to a single integer. Between each pair
   of push moves, the Sokoban can make at most N^2 moves.

   So we can map, say, (P,W) to a single integer P*160000 + W.
   With each push move, we add 160000 and with each walk move we
   add 1 to the cost.

How do we find the shortest path?

1. Do BFS, but maintain unexplored configurations with cost in a
   priority queue (heap), based on cost, rather than in a queue.
   At each step, the next configuration to be explored is the
   minimum cost unexplored configuration in the priority queue.

2. We can label edges between configurations with the cost added
   by this edge. We can then use Dijkstra's algorithm to
   calculate shortest path in M log M time, where M is the number
   of edges, using a heap to store burning times of unburnt
   edges. Note that the number of edges is small --- each vertex
   has degree at most 4, so the total number of edges is about
   2N.

   Notice that if we use a naive implementation of Dijkstra using
   an adjacency matrix, we have to store N*N entries which is too
   large. However, the edges are defined implicitly so we don't
   need to construct an explicit representation.

   As a further optmization, we can use an incremental version of
   Dijkstra's algorithm where we maintain edges in the heap as we
   visit them (see below).

==
=======
Another way of implementing Dijsktra's algorithm

Keep edges you have visited in the heap, adding them as you go
along. The cost stored with an edge is the time at which you
will reach the endpoint of the edge.

Example:

```
          700
        3 ------ 4
    800  /|
      /  | 60
     /   |        500
    1------2 ------ 5 --------- 6
     100    200   \    /
```

```
        100 \   / 50
              7
```

Start at 1. Put 1--800-->3 and 1--100-->2 in the heap.

Pick the minimum, 1--100-->2, and burn 2.

2 has two outgoing edges that are new. Add them to the heap with
values updated by the burn time of 2. So, add 2--160-->3 and
2--300-->5 to the heap.

Note that the costs on the edges in the heap is not the actual
cost but the time at will you will reach the other end of the
edge.

This has the advantage of not adding edges to the heap till you
reach them, which is good if the graph is described implicitly,
as above, where you don't want to explicitly calculate all the
edges.

===
=======

Amazing Robots (IOI 2003)

Two different mazes of size less than 20x20. Each maze contains
a robots (R). Each maze also contains some guards G (at most
10). Mazes may not be the same and robots may not be in the same
starting position.

```
    # # . # . # #      # # . # . # #
    R # . # G . .      . # . . . G .
    . . . . . # #      . . . . . # #
    . G . . . .        . G . . . .
    . . . # . . .      . . . . # R .
    # . # # # . .      # . # # # . .
```

Some cells on the maze are on the periphery, from where the robot
can exit the maze. The aim is to get both robots to exit the
maze without being captured by the guards.

To move the robot, you give an instruction of the form "move(direction)", where direction is one of North, South, East or West. If you say "move(N)", for instance, both robots move one step North, if they can. If either robot is blocked in this direction by a wall, the robot stays where it is. If either robot has left the maze already, the move applies only to the other robot.

The guards move either E-W or N-S on paths of fixed length. You are given the initial direction and the length of his path (this is between 2 and 4). Each time you move the robot, each guard moves one step along his path. If a guard has reached the end of his path, he turns around and moves back. You can assume that the paths followed by the guard are free of blocks, at no time do two guards stand on the same cell.

A guard catches a robot if

-- After some move, the guard and the robot are on the same cell

-- At some move, the guard and the robot cross each other by swapping cells.

Solution:

Do a BFS on the state space.

Naive state space is

1. Position of robots in each grid.
2. Position of each guard.

This is too large (state space is 400 x 400 x ....).

Instead, compute the guards' positions as a function of time. In this case, we need to maintain:

1. Position of robots in each grid.
2. Current time T.

State space is still 400 x 400 x N, where N is the number of steps till they get out, which may become too large.

Observe that the guards' beats are of length 2, 3 or 4. Thus, each guard returns to his starting position after 2, 4 or 6 moves. Thus, we only need to keep track of time T modulo 2, 4 and 6. Since lcm(2,4,6) is 12, if we keep track of T modulo 12, we can recover all these three values.

In this problem, it is important to recognize that the path lengths for the guards allow you to cut down the state space of the BFS dramatically, to bring it within limits. Also, the number of edges is small because each node has at most 4 neighbours, corresponding to the N-S-E-W moves.

==
=======

Post election transfers (IOI Training Camp 2006, Final)

A new government has been elected in Siruseri and has begun the usual process of shuffling all the officials appointed by the previous regime. One of the departments affected is the railways.

The railway network in Siruseri is just a single straight line, with N stations. Of these N stations, L have station masters, distributed at various stations along the line. The government has decided to transfer all station masters so as to keep them as far apart as possible. Unfortunately for the government, there is a clause in the contract negotiated with the railway union by which a station master may be transferred at most K stations from his current posting. The task before the government is to achieve its goal of transferring station masters to maximize the minimum distance between any pair within the restrictions of the contract.

For instance, suppose there are 10 stations, numbered 1,2,..., 10, and there are 3 station masters, currently assigned to stations 5, 7 and 8. Further, let us suppose that no station master can move more than 2 stations away from the current station. Then, the best the government can do is to move the station master at station 5 to station 3 and the station master at station 8 to station 10. In the new arrangement, the station masters are at stations 3, 7 and 10 and the minimum gap is 2

stations, between stations 7 and 10. (Note that, in addition, the station master at station 7 could be moved to station 6, in which case the minimum gap of 2 would be betweens stations 3 and 6.)

Input bounds: L <= 10^5, N and K are integers.

Solution:

Suppose we fix an answer A for the distanct separating the station masters. Can we arrange the station masters if we insist that they are all separated by A?

Let OP(i) be the original position of station master i and let NP(i) be the new position to which we move station master i.

We move the first station master as far left as possible, given that we can shift him by at most K.

  NP(1) = max(1,P(1)-K)

For i+1,

  NP(i+1) = max(NP(i)+A,P(i+1)-K), if NP(i)+A < P(i+1)
       NP(i)+A,   if P(i+1) <= NP(i)+A <= P(i+1)+K
       impossible,   otherwise

Thus, in one linear scan, we can check if a solution exists for a fixed A.

We know that the final solution lies between 1 and N. Since we can solve this problem easily for a fixed A, we can find a good choice of A by binary search by starting with A = N.

This takes time L log N.

===
=======
===
=======
        IOI Training Camp, Lecture Notes, Day 4

                Mon 23 Jun 2008

Computing quantities over intervals in an array

We saw how to efficiently compute the minimum for all M-length
segments of an array, for a fixed value of M. Suppose we want to
efficiently compute the minimum for arbitrary intervals (i,j) in
the array.

We precompute some information, as follows:

```
 2  6  5  8  9  7  11  12  <-- Initial array

 2  6  5  8  9  7  11  12  <-- Minimum in intervals of length 1
                                starting at i
 2  5  5  8  7  7  11      <-- Minimum in intervals of length 2
                                starting at i
 2  5  5  7  7            <-- Minimum in intervals of length 4
                                starting at i
 2                        <-- Minimum in intervals of length 8
                                starting at i
```

Similarly, we can keep track of the minimum in intervals whose
length is a power of 2 ending at i.

```
 2  6  5  8  9  7  11  12  <-- Initial array

 2  6  5  8  9  7  11  12  <-- Minimum in intervals of length 1
                                  ending at i
    2  5  5  8  7   7  11  <-- Minimum in intervals of length 2
                                  ending at i
       2  5  5   7  7  <-- Minimum in intervals of length 4
                              ending at i
                2  <-- Minimum in intervals of length 8
                            ending at i
```

Now, suppose we ask for the minimum value in the range

A[i]...A[j]. Find the largest k such that i+(2^k-1) does not cross
j. The segments A[i]....A[2^k] and A[j-2^k]...A[j] must
overlap. Then,

   min(A[i]...A[j]) is min{ min(A[i]..A[i+2^k]),
                    min(A[j-2^k]..A[j]) }

The two quantities we need are already available in the tables we
have precomputed, so all the work goes into computing the table.

Notice that the first row in the table is immediate --- just copy
the array.

From row i, we can compute row i+1 in linear time --- the minimum
over an interval of 2^k+1 starting at i is the minimum of the two
intervals of size 2^k in previous row starting at i and i+2^k.

Clearly the table has log N rows for an array of size N. We
spend linear time to compute each row, so we can build the table
in time N log N. (Actually, if we only count the values actually
computed, each row is half the length of the previous row, so the
overall time is really only N).

Computing values over intervals in a dynamic array

Suppose we want to report sum(A[i]...A[j]) rather than
min(A[i]...A[j]). We can compute the prefix sums

  P[i] = sum(A[1]...A[i])

for each i, and report

  sum(A[i],A[j]) = P[j] - P[i]

Suppose we additionally allow the array to be updated. We get a
sequence of instructions of the form

   Update(i,a)   ===> update A[i] to A[i] + a
   Sum(i,j)   ===> report the sum from A[i] to A[j]

If we do this naively, we can update the values in constant time,
but we have to actually count the sum from A[i] to A[j] --- we

cannot precompute prefix sums because the values are changing.

We need a new data structure.

Binary index tree

Given an array A, we compute a new array BIT that stores sums of
some segments of A. Here is an example.

```
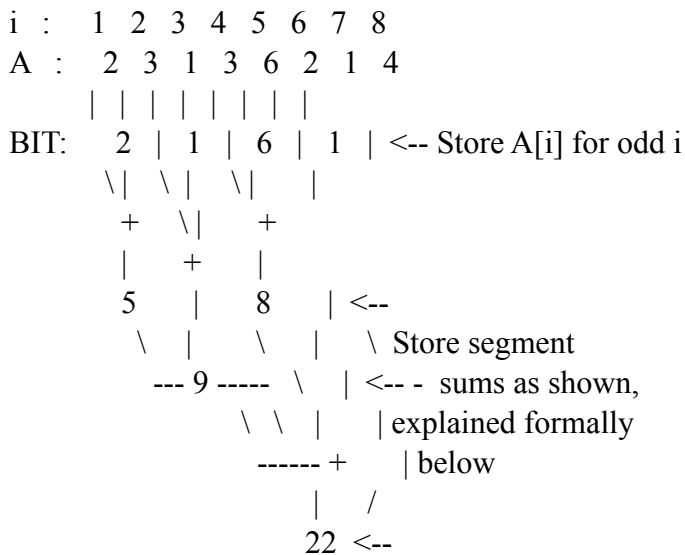i  :   1 2 3 4 5 6 7 8
A  :   2 3 1 3 6 2 1 4
       | | | | | | | |
BIT:   2 | 1 | 6 | 1 | <-- Store A[i] for odd i
        \|  \|  \|    |
         +   \|    +
         |    +    |
         5    |    8    | <--
          \   |    \    |    \ Store segment
           --- 9 -----  \    | <-- -  sums as shown,
                   \ \   |     | explained formally
                    ------ +    | below
                         |   /
                        22  <--
```

What is stored at in BIT[i]?

Write out the positions in binary:

```
i  :   0001 0010 0011 0100 0101 0110 0111 1000
A  :    2    3    1    3    6    2    1    4
```

Let k be the number of trailing zeros in binary representation of
i. In BIT[i], we store the sum of the segment of length 2^k at
position i.

Assuming we can do this, how do we use this data to compute
prefix sums?

Suppose we want the prefix sum A[1] + A[2] + ... + A[i].

We know that the sum of some segment to the left of i is stored
in BIT[i], depending on the binary representation of i.

We can compute the biggest j <= i such that j lies outside this
segment. Call this value next(i). We then pick up BIT[next(i)]
and again compute the biggest k to the left of the segment stored
in BIT[next(i)] etc

How do we compute this value?
$$\overset{<-k->}{}$$
  Suppose i has k trailing 0's. It looks like   x 1 00000

                                 <-k->
  The number we want is i - 2^k. But 2^k is   1 00000.

                                 <-k->
  So, i - 2^k is just   x 0 00000.

So, if we start with i, next(i) can be computed by flipping the
last 1 in the binary represenation of i to a 0. The prefix sum
we want is

   BIT[i] + BIT[next(i)] + BIT[next(next(i))] + ...

We stop when we reach a position that is all 0's.

Each time we go from BIT[i] to BIT[next[i]], we flip one 1 to
0. The binary representation of i has log N bits, so we can
iterate the next() operation at most log N times.

How do we update this data structure when we get an instruction of
of the form Update(i,a)?

We have to update BIT[i] and some BIT[j] for some j > i. Which
values of j are affected?

Clearly, odd values of j are unchanged, since they only refer
to intervals of length 1.
                                 <-k->
Suppose, again that i is of the form x 1 00000.   Suppose we walk
right to position j whose binary representation has r trailing
0's, r < k.

            <-r->

Let j = y 1 00000. The leftvalue value stored in BIT[j] stops short of the value y 0 00000, so this excludes i for sure.

                    <-r->

So, to find the next position j such that BIT[j] includes i, we have to find the next position to the right of i that has at least k trailing zeros. This must be i + 2^k. Again, 2^k is 1 00000, so we add this quantity to i.

  <-k->

Call this value Unext(i). From Unext(i) we go to Unext(Unext(i)) etc. At each step, the number of trailing zeros in Unext(j) is one more than in j, so we only need to update log N entries.

What remains is to compute next(i) and Unext(i) efficiently.

Since next(i) is i - 2^k and Unext(i) is i + 2^i, this boils down to computing 2^k for a given i, where k is the number of trailing zeros.

                            <-k->

We start with i, which is   x 1 00000

                   -    <-k->

If we flip all bits, we get   x 0 11111

                   -    <-k->

Now add 1. We get i' where i'is   x 1 00000

Now, we take a bitwise AND of   <-k->
 i and i' and get   0 1 00000

which is 2^k.

Note that flipping all bits and adding 1 generates the 2's complement, which is the way (-i) is represented in all computers. So, we can obtain 2^k by doing the bitwise AND of i and (-i)!

How do we build the binary indexed tree to start with?

Assume that A consists of all 0's initially, so A[i] = BIT[i] =
0 for all i. Do N updates to insert the initial values of
A[i], 1 <= i <= N, into the tree. This takes N log N time.

===
=======

Mobiles (IOI 2001, Tampere, Finland)

Suppose that the fourth generation mobile phone base stations in
the Tampere area operate as follows. The area is divided into
squares. The squares form an SxS matrix with the rows and columns
numbered from 0 to S-1. Each square contains a base station. The
number of active mobile phones inside a square can change because
a phone is moved from a square to another or a phone is switched
on or off. At times, each base station reports the change in the
number of active phones to the main base station along with the
row and the column of the matrix.

Write a program, which receives these reports and answers queries
about the current total number of active mobile phones in any
rectangle-shaped area.

Solution:

From the previous discussion, it is clear that we need a two
dimensional version of a binary indexed tree.

  BIT[x,y] stores the sum of the rectangle bounded by

        x-2^k....x and y-2^l...y where
          k is the number of trailing zeros in x
          l is the number of trailing zeros in y

    |
    | ((x-2^k)+1,y)   (x,y)
    | o-----------------------o
    | | |
    | | |
    | | |
    | | |
    | | |
    | o-----------------------o

```
|      ((x-2^k)+1,              (x,(y-2^l)+1)
|       (y-2^l)+1)
|
|
  ---------------------------------------------------
```

To compute segment sums, we compute prefix sums with respect to
the starting position. In the two dimensional version, for every
point (x,y) we would like to calculate the sum in the rectangle
from (0,0) to (x,y). Call this sum Rect(x,y).

If we start with BIT[x,y] we get the rectangle above. The next
rectangle to its left is stored in BIT(next(x),y). The next
rectangle below is BIT(x,next(y)). In this way, by walking back
applying next() repeatedly in each coordinate, we can tile the
space from (0,0) to (x,y) with rectangles whose values are stored
in BIT. There are log N rectangles in each dimension, so there
are log^2 N rectangles overall in this tiling.

```
|
|            ((x-2^k)+1,y)        (x,y)
| ... o-----o------------o-------------o
|   | |  |       |             |
|   |  |BIT[x-2^k+1,| BIT[x,y]  |
| ...| ...|  y]     |           |
|   | |  |       |             |
|     o-----o------------------------o (x,(y-2^l)+1)
|   | . |BIT[x-2^k+1,| BIT[x,     |
| ...| . |  y-2^l+1]|   y-2^l+1]|
|     o-----o------------o-------------o
|     ..      ...        ...
  ---------------------------------------------------
```

So, we have

    Rect(x,y) =   Sum   BIT[i,j]}
            i in {x, next(x),...}
            j in {y, next(y),...}

If we update a value A[x,y], we can update all affected values
BIT[x,y] by walking up throught Unext(x), Unext(Unext(x)),... and
Unext(y), Unext(Unext(y)),... independently in each coordinate.

As in the 1 dimensional case, we can build the initial 2-D
indexed tree by starting with 0's everywhere and updating the
initial values one by one. This takes time N^2 (log N)^2.

==
=======

Segment tree [Bentley, 1977]

Another data structure that we can use to solve problems like the
ones above is the segment tree. This can also be used for more
generalized situations, such as when values are updated in an
interval rather than at just one point.

Given an array A, we store the values in the leaves of a tree.
At each level we combine two nodes at time and store their sum in
the parent. This builds up a binary tree, as shown in the
following example with an array of size 8.


```
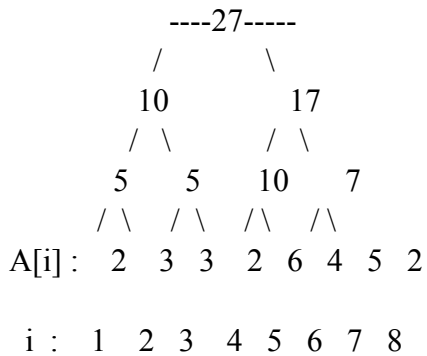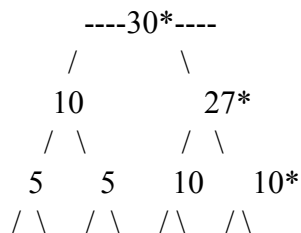             ----27-----
            /           \
          10             17
         /  \           /  \
        5    5        10    7
       / \  / \      / \   / \
A[i] : 2  3 3  2    6  4  5  2
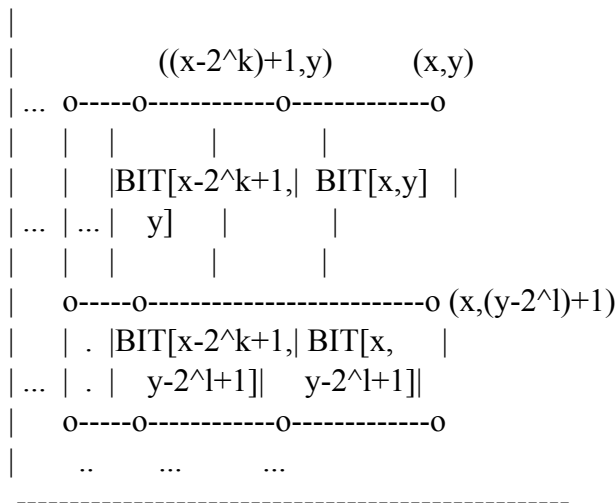
 i :  1  2  3  4   5  6  7  8
```

Now, suppose we update A[7] from 5 to 8. To update the tree, we
start from the leaf A[7] and walk up a path of length log N to
the root, updating all values on the way. In the figure below,
the updated values are marked with a *.

```
             ----30*----
            /           \
          10             27*
         /  \           /  \
        5    5        10    10*
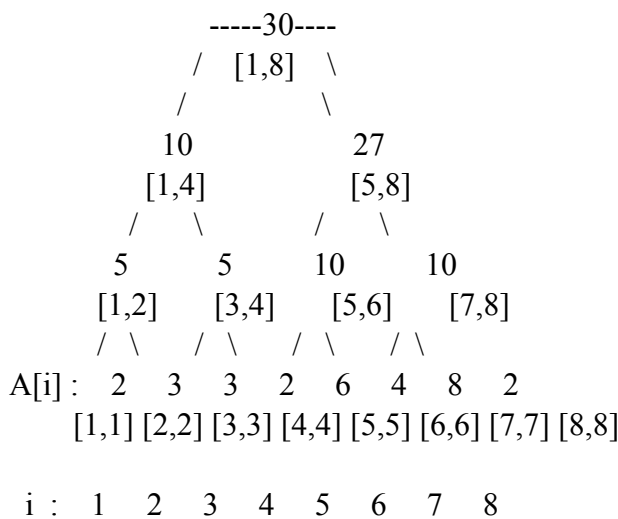       / \  / \      / \   / \
```

A[i] : 2 3 3 2 6 4 8* 2

 i : 1 2 3 4 5 6 7 8

Given this data structure, how do we find the prefix sum P[i]?

At each node, keep track of what segment it stores.

```
            -----30----
           /   [1,8]   \
          /             \
        10               27
      [1,4]             [5,8]
      /   \            /    \
     5      5        10       10
   [1,2]  [3,4]     [5,6]    [7,8]
   / \    / \       / \      / \
A[i] :  2   3   3   2   6   4   8   2
   [1,1] [2,2] [3,3] [4,4] [5,5] [6,6] [7,7] [8,8]

 i :  1   2   3   4   5   6   7   8
```

Suppose we want the value P[6]. We start at the root and observe
that [1-6] is contained in [1-8]. Going left we find [1-4] that
is contained in [1-6] but not all of it. What is left is [5-6]
for which we go right. Seeing [5-8], we go left and find [5-6]
and stop.

What if we want P[7]? Again, we pick up [1-4] and go to [5-8].
We then pick up [5-6] and go right to pick up [7-7].

In general, when computing P[j], we pick up the value of the left
child of j and, if any part of the segment [1-j] is still
remaining, we go right. At each step we pick up the entire left
child and go right. This can take at most log N steps.

How do we store this tree? We can number the elements in the
tree level by level, starting from the root, as we do in a heap.

If there are N leaves, there are 2N-1 elements in the array, with
the leaves in the last N positions. Given a position i, we can
compute the positions parent(i), leftchild(i) and rightchild(i)
as in a heap.

To set up the array, do something similar to constructing a heap.
The array elements are stored in the last N elements. Working
backwards, update A[i] as A[2i]+A[2i+1] (sum of its children).

As we have seen before, given the prefix sums, we can compute
sum(i,j) for any segment A[i]..A[j] by the expression P[j]-P[i].

Suppose we want, instead, to compute the minimum value for
arbitrary segments.

Again we build the tree by keeping the minimum of each interval.


```
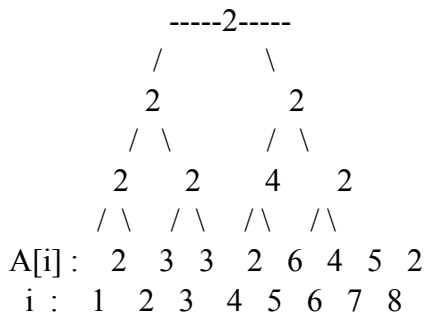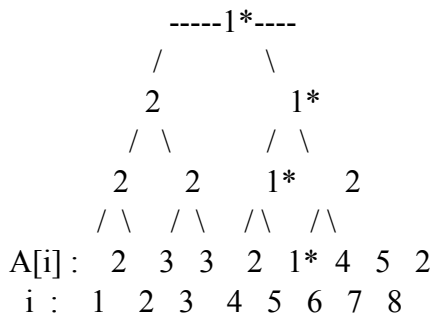              -----2-----
             /           \
            2             2
          /   \         /   \
         2     2       4     2
        / \   / \     / \   / \
A[i] :  2   3   3   2   6   4   5   2
  i  :  1   2   3   4   5   6   7   8
```


If we update a value, we propagate the update up to the root as
before. For example, suppose we reset A[5] to 1.

```
              -----1*----
             /           \
            2             1*
          /   \         /   \
         2     2       1*    2
        / \   / \     / \   / \
A[i] :  2   3   3   2   1*  4   5   2
  i  :  1   2   3   4   5   6   7   8
```

How do we find min(i,j) for an arbitray interval A[i]...A[j]?

We start at the root and split (i,j) to the part that goes left

and right.

For instance, to compute min(2,5), we split the interval [2..5] as [2..4] and [5..5] and send the first search left and the second one right. On the left, we further split [2..4] as [2..2] and [3..4] etc.

Why does this work in log n time?

We have an overall interval [1..n], say [1..8] and an interval [i..j] to decompose. If [i..j] is entirely contained in [1..n/2] or [n/2+1..n], we don't have to break it up at the root.

The interesting case is when [i..j] spans the midpoint. Then, the interval splits as [i..n/2] and [n/2+1..j]. Notice that now we have an interval in each subtree such that one endpoint is the end of the overall interval in that subtree (the interval we are search for "sticks" to the right end or the left end of the overall interval).

Now, if we have a node with an interval [k..l] and we are looking for an interval [k..m] sticking to its left, we either do not branch at all (if m < midpoint) or we split, but the left path stops at the next level.

For instance, initially [2,6] splits at [2,4] and [5,6]. 4 is the right end point of [2,4] and 5 is the left endpoint of [5,6]. When [2,4] next splits, it becomes [2,2] and [3,4]. Here, [3,4] is a complete interval and need not be further decomposed: just pick up the value at the internal node [3,4].

Hence, computing min(i,j) takes only O(log N) time.

===
=======

From inversions to permutations

Given a permutation that maps each i to P(i), Inversion(i) is the number of j < i that appear to the right of i in the permutation. We can thus associate with each permutation, an

inversion sequence. An example is given below.

```
                1 2 3 4 5 6 7 8
Permutation P(i)   2 3 1 6 7 8 4 5
Inversion  I(i)    0 1 1 0 0 2 2 2
```

How do we recover the permutation from the inversion sequence?

1. Start with largest number, here 8. I(8) = 2, so there are two
   blank space to the right of 8, so we place 8 in position 6.

```
          1 2 3 4 5 6 7 8
          - - - - - 8 - -
```

2. Next, omit the position where 8 has been placed and examine
   I(7) with respect to the remaining seven positions. Since
   I(7) = 2, 7 has two elements to its right among the seven
   blank, and hence it goes in slot 5 (slot 6 is already filled
   by 8).

```
          1 2 3 4 5 6 7 8
          - - - - 7 8 - -
```

3. Omit the positions where 8 and 7 have been placed and examine
   I(6). We need to leave two blank, so it goes in slot 5 (slots
   5 and 6 are already filled by 7 and 8).

```
          1 2 3 4 5 6 7 8
          - - - 6 7 8 - -
```

3. Omit the positions where 8, 7 and 6 have been placed and examine
   I(5). We need to leave zero blank, so it goes in the last slot.

```
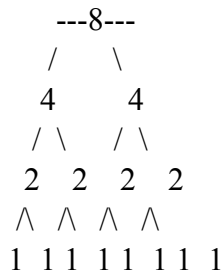          1 2 3 4 5 6 7 8
          - - - 6 7 8 - 5
```

   ...

   In general, at each step, use I(j) to determine how many blank
   slots to leave empty at the right of the current set of slots.

Complexity:

In each pass, we have to scan the array to find the kth blank slot remaining. This takes O(N) time, so overall, this takes O(N^2) time.

Can we do this in time N log N using a segment tree?

We maintain a segment tree in which we maintain the number of blanks to the right in each interval. Initially, all leaves are empty and the tree looks like this.

```
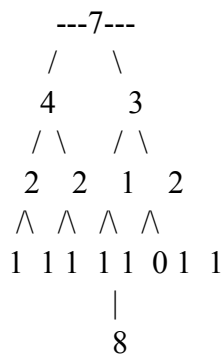      ---8---
      /    \
     4      4
    / \    / \
   2   2  2   2
  /\  /\  /\  /\
  1 1 1 1 1 1 1 1
```

Now consider I(2) = 1. This tells us that 8 must go in the 3nd blank leaf from the right, or, equivalently, the 6th blank leaf from the left since the value at the root tells us there are totally 8 blank positions available. We walk down to this leaf, set the number of blanks in the interval [6,6] to 0 and propagate the change up.

```
       ---7---
       /    \
      4      3
     / \    / \
    2   2  1   2
   /\  /\ /\  /\
   1 1 1 1 0 1 1
             |
             8
```

Now, we examine I(7) = 2. We need to put it in the 3rd position from the right or, since the root reports 7 blank positions, we need the 5th blank position from the left. The root has only 4 blank leaves on the left, so we have to go right and look for the (5-4) = first blank leaf on the right. Next, walk left and left and, updating intermediate values, we get.

```
      ---6---
      /    \
```

```
     4      2
    / \    / \
   2  2  0  2
  /\ /\ /\ /\
 1 1 1 1 0 0 1 1
      | |
      7 8
```

...

In this way, we can insert each number in log N steps into its
correct position in the permutation. There are N numbers to
insert, so this takes N log N steps.


```
======================================================================
=======
```
Session 2, 11:00--12:30
```
======================================================================
=======
```
Paint (introductory version)

So far, we have considered an array in which we make updates at
single positions and ask questions about intervals. What if we
have, instead, interval updates and point queries?

A painter paints a strip N squares wide


```
        ----------------------
        | 1 | 2 | 3 | .... | N |
        ----------------------
```

In one step, the painter paints a continuous strip from square i
to square j, which we denote [i,j]. Periodically, we want to
know the number of layers of paint that have been applied at
square p.

Assume we have a sequence of instructions of the form:

paint([i1,j1])
paint([i2,j2])
.
.

paint([ik,jk])
layers(p)

There are K paint operations and M queries.

Naive solution:

Initialize layers(p) = 0 for all $1 <= p <= N$.

On each paint([i,j]), increment layers(p) for each $i <= p <= j$

In the worst case, this could take O(N) time for each paint operation. A query can be answered in constant time. Thus, this solution takes time O(KN).

Better solution:

Maintain a segment tree, but what do we store at each node?

Recall that each node represents an interval. For N = 8, here are the intervals spanned by the nodes in the tree.

```
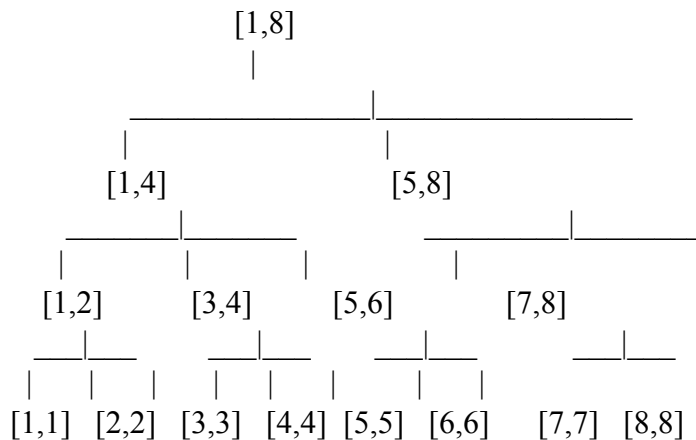                    [1,8]
                      |
         _____|_____
         |                     |
       [1,4]                 [5,8]
     _____|_____       _____|_____
     |           |       |           |
   [1,2]       [3,4]   [5,6]       [7,8]
   __|__       __|__   __|__       __|__
   |   |       |   |   |   |       |   |
 [1,1] [2,2] [3,3] [4,4] [5,5] [6,6]  [7,7] [8,8]
```

At each vertex v, record

count(v) = number of paint strokes that contain the entire
interval Interval(v), but do not contain the entire
interval Interval(parent(v)) spanned by the parent
of v.

How do we update this?

Consider an operation paint([i,j]):

If [i,j] = [1,N], update count of the root node and stop. For every other node, Interval(v) is contained in [i,j] but so is Interval(parent(v)), so count(v) is unchanged.

If [i,j] is not equal to [1,N], break up the interval according to the children.

For instance, in the example above, suppose [i,j] = [2,5]. This does not span [1,8] so we split [2,5] as [2,4] and [5,5], send [2,4] down the left tree and [5,5] down the right tree.

We repeat this recursively at these two children of the root using the same rule. Thus, on the left child of the root, since [2,4] does not span [1,4], we further split it as [2,2] and [3,4] and send [2,2] left and [3,4] right. [3,4] stops, but [2,2] goes down further to the leaf labelled [2,2].

We update the count(v) for every v where the (sub)interval of [i,j] that reaches v matches Interval(v).

As before, we can argue that each update takes time log N. Thus, over K paint operations, updating the entries takes O(K log N) time (as opposed to O(KN) time in the naive solution).

How do we answer a query of the form layers(p)?

We just have to add up count(v) for all nodes on the path from the root to [p,p]. How do we do this efficiently? Recall that we have stored the tree in an array of size 2N-1. The leaf nodes are in positions N to 2N-1. Thus, the leaf node corresponding to the interval [P,P] is at position (N-1)+P. We can start here and walk up to the parent till we reach the root.

Alternatively, we can start at the root and walk down the path to [P,P] by checking the current interval and deciding whether to go left or right at each step.

This requires O(log N) steps for each query, or O(M log N) steps for M queries.

Thus, overall, this takes $O((K+M) \log N)$ steps to update information and answer queries. If K,M and N are roughly equal, this takes $O(N \log N)$ time whereas the naive solution takes $O(N^2)$ time.

Question to think about.

What if we change our queries to ask:

Minlayers([i,j]) --- among all squares in the interval [i,j],
                     report the minimum number of layers among
                     the squares in this interval

===
==

Trees

A tree is a connected graph without cycles.

```
           o-----o
          / \
   o-------o   o-------o
       /  |\
      /   | \
     o    o o
```

Claim: In any tree there is a unique path from one node to
       another. Verify this for yourself.

Trees have some nice properties.

For instance: If we do dfs(v) from any vertex v, the resulting
  dfs tree will be isomorphic to the original tree because there
  are no back edges.

Exploring a tree from a fixed vertex is called "rooting the tree"
at that vertex. Having rooted the tree, the furthest nodes
become leaf nodes.

For many properties, we can then compute information about the
tree in a natural bottom-up order, starting from the leaves.
Once we have information for all children of a node, we can
update the value at that node.

Claim: A tree with n vertices has exactly n-1 edges. (Why?)

Check: If a graph is connected and m = n-1, it must be a
       tree.

==
=======

Catering Contracts: Two Way Split

Railway stations organized as a tree. Each station has a canteen
with an expected profit. Contracts have to be given to two
caterers such that each caterer gets a contiguous set of
stations. We want to allocate all the canteens to these two
caterers such that the difference in total expected profit
between the two is minimized.

Example

```
            a 3
            |
            b 4
          / \
         /   \ 7
      6 c     g ----h 9
        |     |
      4 d     f 8
          / \
        3 e     i 4
```


Solution

We have to divide the tree into two connected subgraphs. Notice

that each subgraph remains a tree. Recall that a graph on n
nodes is a tree iff it has n-1 edges. If we split the graph up
as n1,n2 such that n1+n2 = n, we must have n1-1 edges on the left
and n2-1 edges on the right, so the overall number of edges in
the two subgraphs is (n1+n2)-2 = n-2. Originally, we had a tree
with n nodes and n-1 edges, so dividing the tree into two parts
is the same as dropping one edge from the original tree.

We explore all choices of edges to drop. For each choice of edge
to drop, we compute the difference and, overall, we minimize the
difference. Notice that the sum of the two parts is always equal
to the sum of the whole network, so minimizing the difference is
the same as minimizing the maximum.

Brute force:

Pick any vertex in the tree as the root. Do a DFS and add up the
weight of the whole tree. Now, delete an edge and repeat the DFS
from the same root to get the weight of first component.
Subtract this from total weight to get overall component. To do
this, we have to repeat the calculation for each deleted edge.

More clever:

We root the network at some station to get a tree. (How do we
root the tree? Pick a vertex and do a DFS from that node. We
will actually not construct the tree but do a DFS like
computation that implicitly follows the DFS tree.)

```
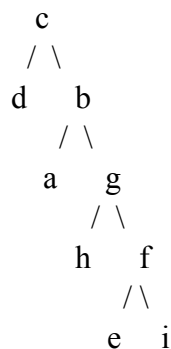         c
        / \
      d    b
          / \
        a    g
            / \
          h    f
              /\
            e    i
```

For a vertex v, let Sub(v) be the sum of the profits of the
subtree rooted at v

```
        48
         c
     4  / \ 38
       d    b
          / \ 31
        3 a    g
            / \   15
          9 h    f
                /\
              e   i
             3    4
```

We can compute Sub(v) using a modified DFS as follows.

```
mdfs(v){
  sum = Weight(v);
  foreach unvisited neighbour w of v{
    sum = sum + mdfs(w);
    mark(w);
  }
  Sub(v) = sum;
  return (sum);
}
```

During the DFS, we have calculated at each node w the total
weight Sub(w) of the subtree rooted at that node. Suppose the
edge we want delete is (u,v). Depending on whether u is a parent
of v or vice versa in the tree, either Weight(u) or Weight(v)
gives the total value of in one component split by (u,v). Since
we know the total, we can subtract this value from the total to
et the value of the other component.

For any edge u-v, how do we decide whether u is the parent of v
or vice versa. We compute the DFS numbers of the nodes while
doing mdfs. Among u and v, the one with the smaller DFS number
is the parent.

This will take linear time --- there are N nodes, N-1 edges.

===
=======

Catering Contract: Three Way Split (IARCS Online Competition)

Railway stations organized as a tree. Each station has a canteen
with an expected profit. Contracts have to be given to three
caterers such that each caterer gets a contiguous set of
stations. We want to allocate all the canteens to these two
caterers so that the maximum profit across the three caterers is
minimized.

Solution:

For each edge, remove one edge and do a two way split on both
sides. This will take time N^2.

==
=======

Catering contracts

We have a railway network with no cycles with expected profit at
each station. Contractors bid for a subset of stations to
maximize profit. A caterer may not bid for two adjacent
stations. What is the best way to pick the stations?

For instance, here is a network:


```
      10              20
       \            /
        30--------------40
       /            \
      20              30
```

In this case, the best choice are the outer four nodes with value
10+20+20+30 = 80.

Solution:

We know the network is a tree. How can we use this fact to evaluate the best profit?

We root the network at some station to get a tree. For a vertex v, let $T(v)$ denote the subtree rooted at v and $P(v)$ denote the maximum profit in the stations in $T(v)$.

$$P(v) = \max \{ \quad \underset{\text{w child of v}}{\text{Sum}} \quad P(w), \qquad \text{// we exclude v}$$

$$\underset{\text{w grandhild of v}}{\text{Sum}} \quad P(w) + V(v) \quad \text{// we include v}$$
$$\}$$

An alternative formulation

$P1(v) = \max$ in $T(v)$ including v
$P2(v) = \max$ in $T(v)$ excluding v

$$P1(v) = V(v) + \underset{\text{w child of v}}{\text{Sum}} \quad P2(w)$$

$$P2(v) = \underset{\text{w child of v}}{\text{Sum}} \quad \max(P1(w),P2(w))$$

The second formulation more explicitly captures the case when we include and exclude v and is probably clearer in terms of understanding the structure of the problem.

How do we program this?

Do a dfs like computation:

Version 1: With global arrays:

```
eval(v){  /* updates global arrays P1 and P2 */
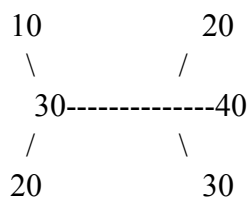
   Mark v as visited;

   P1[v] = V(v);  /* Initialize both values */
   P2[v] = 0;
```

```
    if (v is a leaf){
      return;
    }


    for all unvisited neighbours w of v
    {
      eval(w);
      P1[v] += P2[w];
      P2[v] += max(P1[w],P2[w]);
    }
}
```

Version 2: Each call to eval returns pair (P1(v),P2(v))

```
 eval(v){  /* returns ((P1(v),P2(v)) */

    Mark v as visited;

    if (v is a leaf){
      return(V(v),0);
    }


    for all unvisited neighbours w of v
    {
      (p1,p2) = eval(w);
      myp1 += p2;
      myp2 += max(p1,p2);
    }

    return(p1+V(v),p2);
  }
```

==
=======
Rods (IOI 2002)
----

1. One horizontal rod and one vertical rod is placed on a
   (hidden) N x N grid.

2. The only way to find out information about the rods is to call
   a library function rect(x1,y1,x2,y2), x1 < x2, y1 < y2.

   This function returns 1 if any part of either rod lies within
   the rectangle defined by top left corner (x1,y1) and (x2,y2),
   0 otherwise.

3. Make as few calls as   (x1,y1)
   possible to the library   . . . . _____ . . .
   function. Your score   . . . . | . . . . | . . .
   depends on the number of   . . . x | . . . . | . . .
   calls you make to rect(...).   . x x x x x x x x | . . .
                           . . . x | . . . . | . . .
   For example, in the   . . . x |_____| . . .
   picture to the right,   . . . x . . . . . (x2,y2)
   the call to   . . . x . . . .
   rect(x1,y1,x2,y2)   . . . .
   returns 1.   . . . .

Solution suggested in class:
----

1. Starting with the full rectangle, use binary search to narrow
   down to a single "x".

2. Probe the four directions around this "x" to check whether it
   is a vertical or horizontal rod (or a crossing point of two
   rods).

3. Having got the orientation, do a binary search along that line
   (use rectangles in which one dimenstion is 1) to find the end
   points.

4. Assume that the rod we have found is the horizontal one.
   Query the rectangles (strictly) above and below the rod to
   find an "x" from the vertical rod.

5. Again, use binary search to narrow it down to a single "x" and
   then use binary searches along the vertical line containing
   this "x" to find the end points.

Official Solution:
----

1. Find the bounding rectangle (smallest rectangle containing both rods) using 4 binary searches. One binary search to find leftmost column with an x, one more to find rightmost column with an x, one more for topmost row with an x, one more for bottommost row with an x.

2. Within the bounding box of size P x Q, make 4 queries about the corners to find out the way the rods are arranged (cases shown below upto rotation/reflection).

```
3 corners      Two adjacent      No corners   Two opposite
are 1          corners are 1     are 1        corners are 1


xxxxxxx       xxxxxxxxxx         x         xxxxxx
x             x             xxxxxx              x
x             x                 x          x
x             x                 x          x


  OR            OR


xxxxxxx       xxxxxxxxxx

 x            x
 x            x
 x            x
```

3. Now, with two additional binary searches, you can find out the actual layout of the rods.

===
=======

XOR (IOI 2002)

A N x N grid of pixels (can be set to white or black). At each step, specify we can make a call XOR(x1,y1,x2,y2). This selects a rectangle of pixesl with top left corner (x1,y1) and bottom right corner (x2,y2) and flips all the values in this rectangle (negation is the same as XOR with 1/black, hence the name).

For example (. denotes white, X denotes black)

```
  1 2 3 4 5 6 7 8 9              1 2 3 4 5 6 7 8 9
1 . . . . . . . . .          1 . . . . . . . . .
2 . . X X . . . . .          2 . . . . . . . . .
3 . X . . X . . . .          3 . X X X X . . . .
4 . X . . X . . . .  XOR(2,3,5,4)  4 . X X X X . . . .
5 . . X X . . . . .  ===========>  5 . . . . . . . . .
6 . X . . X . . . .          6 . X . . X . . . .
7 . X . . X . . . .          7 . X . . X . . . .
8 . X . . X . . . .          8 . X . . X . . . .
9 . . X X . . . . .          9 . . X X . . . . .
```

Given an initial configuration, make the minimum number of XOR
calls to make the pixel grid entirely white.

The instance above can be solved in 3 calls to XOR.

```
  1 2 3 4 5 6 7 8 9              1 2 3 4 5 6 7 8 9
1 . . . . . . . . .          1 . . . . . . . . .
2 . . X X . . . . .          2 . . . . . . . .
3 . X . . X . . . .  XOR(2,3,9,4)  3 . X X X X . . . .  2 moves
4 . X . . X . . . .  =========>   4 . X X X X . . . .  ==>
5 . . X X . . . . .          5 . . . . . . . . .
6 . X . . X . . . .          6 . X X X X . . . .
7 . X . . X . . . .          7 . X X X X . . . .
8 . X . . X . . . .          8 . X X X X . . . .
9 . . X X . . . . .          9 . . . . . . . . .
```

Solution:

To be discussed later.

==
=======
==
=======
          IOI Training Camp, Lecture Notes, Day 5

                    Tue 25 Jun 2008
==
=======
Session 1, 09:00--10:30
==
```

========
Basics of graphs

G = (V,E)   V : Vertices
        E : Edges, subset of V x V
                (V x V is set of all pairs of vertices)

We can draw a graph by representing vertices as points and edges
as lines joining pairs of points. Here are two examples of
graphs:

```
   o-----o        o
   |   \     / \
   |    o    o  o---o
   |   /     \ /
   o-----o        o
```
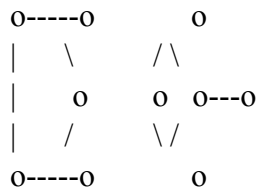
By convention, n = |V| = number of vertices, m = |E| = number of
edges.

We normally assume we work with "simple graphs"
-- no self loops
-- only one edge between a pair of vertices ("end points of the
   edge")

Degree of a vertex v:

  Number of edges with v as one end point

  The sum of the degrees of a graph is even. Each edge adds 1
  to the degree of the vertex at both end points. Thus,

    sum   degree(v) = 2 |E| = 2m
   v in V

  where |E| denotes the size of the set of edges E.

Path:

  A sequence v_1 v_2 ... v_k of distinct vertices such that each
  adjacent pair v_i v_i+1 is an edge. (Note: a path cannot

visit the same vertex twice. If a vertex is visited more than once in such a sequence, it is called a walk, not a path.)


Cycle:

  A sequence v_1 v_2 ... v_k v_1, where v_1 v_2 ... v_k is a
  path and v_k v_1 is an edge. (i.e. a cycle is a path that is
  closed by an edge from the last vertex in the path back to the
  initial vertex.)

Connected:

  u and v are connected if there is a path from u to v. A graph
  is connected if every pair of nodes is connected.

Tree:

  A graph without cycles (in an undirected sense) with exactly
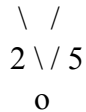 one component, that is, the graph is connected.

```
        o-----o
        / \
   o-------o   o-------o
      /   |\
     /    | \
    o     o o
```
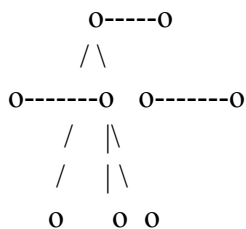
  Can think of a tree as a minimal connected graph.

  - By induction, can show that a tree on n vertices always has
    exactly n-1 edges.
  - In any tree, there is a unique path between any pair of
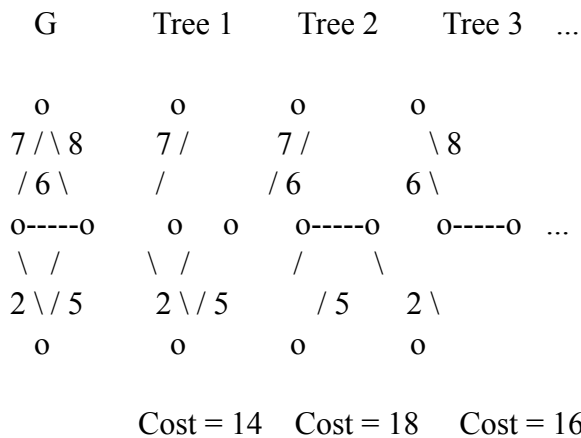   vertices.

Weighed graph:

  Attach a cost with edge

```
     o
    7/\ 8
    / 6 \
   o-----o
```

```
  \  /
 2 \ / 5
    o
```

Spanning tree

Let G = (V,E) be a connected, undirected, weighted graph. Find a
tree that "touches" all the vertices, using a subset of the edges
of the original graph. This is called a spanning tree.

A graph could have more than one spanning tree.

```
    G         Tree 1      Tree 2      Tree 3   ...


    o           o           o           o
  7 / \ 8     7 /         7 /            \ 8
  / 6 \       /           / 6           6 \
  o-----o     o   o       o-----o      o-----o  ...
  \  /        \  /        /    \
 2 \ / 5      2 \ / 5      / 5     2 \
    o           o           o           o

              Cost = 14   Cost = 18   Cost = 16
```

How do we find the minimum cost spanning tree?

Solution:

Sort weights in ascending order --- in the example above,
2,5,6,7,8.

Include the edges into the spanning tree in ascending order.
Start with 2, add 5. Next is 6. This creates a cycle, so we
cannot add it. (To detect whether an edge creates a cycle, mark
each vertex that has been included in the tree. If the next edge
has both end points marked, it will form a cycle). So we skip 6,
add 7. Now we have m-1 edges and we stop with the spanning tree
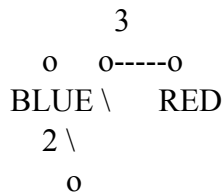Tree 1 above.

This may not always work so smoothly. When we pick up the next
smallest edge, neither end point may be marked already. Thus, we

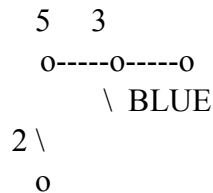have an intermediate stage that is a forest, rather than a tree.

```
     G


      o
    7 /\ 8
     / 5 \  3
    o-----o-----o
     \  /
    2 \/ 6
      o
```
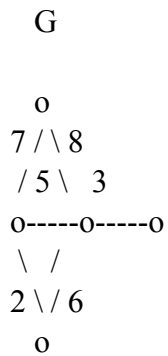
Here the edges in sorted order are {2,3,5,6,7,8}. We start with
2 and add 3. These two edges form disjoint components (a
forest). If we now consider 5, we can add it though both edges
are marked. We use different "colours" to mark each component.
An edge that connects marked nodes of different colour can be
added --- the components get merged, so convert one colour to the
other to signify a new single component.

After adding {2,3}   After adding {2,3,5}

```
         3              5    3
    o    o-----o          o-----o-----o
   BLUE \    RED               \ BLUE
     2 \                  2 \
       o                    o
```

Implementing this algorithm requires a good data structure to
keep track of component, merge them etc. We will see this
later.

This is called Kruskal's algorithm.

Complexity:

1. Sorting takes O(m log m) time
2. We have to examine, in the worst case, all m edges. To stay
   within the sorting cost, the set operations should work in
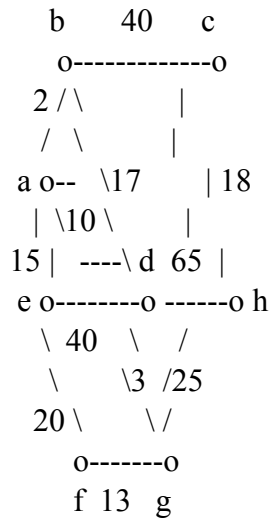   log m time.

Prim's algorithm

Grows the tree, rather than constructing a forest and combining components.

Start with a edge of minimum weight. This edge should be in the final tree and will be connected to the rest of the tree at (at least) one of its end-points. Choose the smallest outgoing edge from either end point.

At any stage, we have a set of vertices S that are already connected as a tree by a set of edges T.

To add a new vertex to S, pick the smallest edge (u,v) from u in S to v outside S. Add v to S and (u,v) to T.

Example:

```
            b      40     c
              o-------------o
            2 /\            |
             /  \           |
          a o--  \17       | 18
            | \10 \         |
          15 |  ----\ d  65 |
            e o--------o ------o h
              \ 40    \   /
               \       \3 /25
            20 \        \ /
                 o-------o
                f  13  g
```


   Edge (a,b) of length 2 is minimum overall edge, so set

     S = {a,b}
     T = {(a,b)}

Now we have to consider all edges out of S. (In the picture, edges in T are labelled %, vertices in S are labelled *).

```
      b     40    c
       *-------------o
      8 % \
      %   \
    a *--  \17
    | \10 \
   15 |  ----\ d
    e o     o
```

Smallest outgoing edge from S is (a,d), so

S = S U {d} = {a,b,d}
T = T U {(a,d)} = {(a,b),(a,d)}


```
       b     40    c
        *-------------o
       8 % \
       %   \
     a *%% --\17
     | % 10\
    15 |  %%%%\ d  65
     e o--------* ------o h
        40   \
              \3
               o
               g
```
...

Does it matter that we start with the shortest edge?

 It does not matter. Why?

Implementation:

 The code is a variant of Dijkstra's algorithm. At each stage,
 when you burn a vertex u, you update the expected burn time of
 each unburnt vertex v as the minimum of

  - current shortest edge from burnt vertices to v
  - cost of the edge (u,v)

 The difference from Dijkstra's algorithm is that we don't

incorporate the burn time of u in the cost.

Again, we can use a heap to calculate the minimum unburnt vertex.   In the heap, we would keep the minimum cost edge from the burnt vertices to each unburnt vertex. (We only need to keep the smallest currently known edge to each unburnt vertex.)

The complexity analysis is similar to Dijkstra.

===
=======

Maintain (IOI 2003)

Farmer John's cows wish to travel freely among the N (1 <= N <= 200) fields (numbered 1...N) on the farm, even though the fields are separated by forest. The cows wish to maintain trails between pairs of fields so that they can travel from any field to any other field using the maintained trails. Cows may travel along a maintained trail in either direction.

The cows do not build trails. Instead, they maintain wild animal trails that they have discovered. On any week, they can choose to maintain any or all of the wild animal trails they know about.

Always curious, the cows discover one new wild animal trail at the beginning of each week. They must then decide the set of trails to maintain for that week so that they can travel from any field to any other field. Cows can only use trails which they are currently maintaining.

The cows always want to minimize the total length of trail they must maintain. The cows can choose to maintain any subset of the wild animal trails they know about, regardless of which trails were maintained the previous week.

Wild animal trails (even when maintained) are never straight. Two trails that connect the same two fields might have different lengths. While two trails might cross, cows are so focused, they refuse to switch trails except when they are in a field.

Solution:

Given G and an MST T for G. Add an edge e with weight w(e) to
G. Find MST for G+e.

1. Recompute MST --- O (m log m)

2. Find MST on T+e --- O(n log n)

Alternatively

3. Include e and delete heaviest edge in cycle formed by e in
   T+e --- O(n)

What if we delete an edge from G and look for MST in G-e?

1. Recompute MST --- O (m log m)

2. If e is not in T, do nothing. If e belongs to T, delete e,
   which separates T into a cut (U,V). Include smallest weight
   edge between U and V. O(n^2).

==
=======

Directed acyclic graphs (dags)

Directed graphs:

  Each edge has a direction: one end point is the source and the
  other end point is the target.

  If G is directed, we could have an edge from v -> w and
  another from w -> v. This does not violate our assumption of
  simple graphs.

  For directed graphs, we talk of "outdegree" and "indegree" of
 a vertex instead of just degree.

  We can "forget" the orientation of edges and obtain the
  underlying undirected graph.

A directed graph that does not have any (directed) cycles is
called a directed acyclic graph (dag).

Sorting a dag

List out the vertices in a sequence so that there is no dag edge
going from any vertex in the sequence to an earlier vertex in the
sequence.

Here is a dag (all edges oriented left to right):

```
    -->d------->c
   /     /   \
  a------->b    -->g
   \  /  \  /
    -->e------>f
```
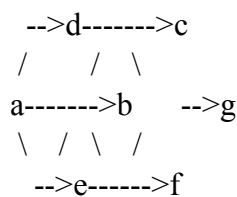
Here is a sorted sequence of the type we want:

  a e d b c f g

This is called a "topological sort" of the dag. In general,
there will be more than one such sequence. Formally, here is how
topological sort works.

In a dag, there must be at least one vertex with no incoming
edge, or indegree 0. (Otherwise, we can keep moving backwards.
There are only n vertices overall. After we traverse n edges, we
must hit a vertex we have seen before, which means there is a
cycle, so the graph is not a dag.)

A source vertex is a possible starting point of a longest path.
(A non-source vertex cannot be the starting point of a longest
path because we can extend this path by adding of the vertices
that point into it.)

We could have multiple source vertices. We classify all these as
level 0. The neighbours of level 0 vertices are in level 1.
This is similar to doing a simultaneous BFS from all the source
vertices.

In other words, we are ranking vertices as follows:

rank(u) = 0 if u is a source vertex
rank(u) = i if the longest path from some source vertex to u
         is of length i

To compute the rank, we maintain a queue, as in BFS.

```
/* indegree(v) is the number of edges that enter v */

compute indegree(v) for all v;

/* First process source vertices */
for each vertex v with indegree(v) = 0{
  rank(v) = 0;
  add v to the tail of in a queue;
}

/* Process remaining vertices */
until the queue is empty{
  remove the vertex v at the head of the queue;
  delete v from the graph;
  recompute indegree(w) for all w;

  /* Check if we have found a vertex all of whose
     predecessors have been processed */
  if indegree(w) becomes 0{
     rank(w) = rank(v) + 1;
     add w to the tail of the queue;
   }
  }
}
```

How do we (re)compute indegree? Initially, we have to compute
all edges. We have to take each pair of words and check if they
are at edit distance 1. The edit check can be done in time
linear to the lengths of the words. If we neglect this,
computing the edge relation is $O(n^2)$. Along with this, we can
compute indegree initially in time $O(n^2)$

When we recompute the indegree after deleting a vertex v, we only
have to decrement indegree(w) for each outgoing neighbour of v.
So, we make an indegree update once for edge in the original dag.

Hence, a more direct way of coding "delete a vertex and update indegrees of all remaining vertices" is as follows:

```
  /* Process remaining vertices */
  until the queue is empty{
    remove the vertex v at the head of the queue;
    for each edge (v,w){
      decrease indegree(w) by 1;
      if indegree(w) becomes 0{
        rank(w) = rank(v) + 1;
        add w to the tail of the queue
      }
    }
  }
```

===
==

Edit ladder

Given a sequence of words w_1, w_2, ..., w_n, two words u and v are neighbours if they have edit distance 1 (that is, u can be transformed to v by changing one letter, adding one letter or deleting one letter).

e.g. cat--cot

    rat--rot--rob

We orient these edges in dictionary order : can go from cat to cot but not reverse. This defines a directed graph.

    cat-->cot

    rob--
        \
    rat----->rot

We want to find the longest path in this graph.

Solution:

Construct a graph whose nodes are the words w_1, ... w_n
There is an edge from w_i to w_j if i < j and w_i and w_j are
neighbours.

Want the length of the longest path in this graph. In general,
finding the longest path in a graph efficiently is difficult. No
algorithm short of brute force enumeration of all paths is known.

However, observe that this graph is a dag. There are no cycles
because any edge goes from left to right in the list of words.
For a dag, we can calculate the longest path by doing a
topological sort and finding the maximum rank.

Why does this give a longest path? Observe that any vertex v at
rank k+1 must have at least one incoming edge from a vertex at
rank k --- otherwise, this vertex v would have got a lower rank.
Thus, for every vertex at rank n, we can walk backwards in the
graph edge by edge, decreasing the rank by one each time, till we
reach a vertex of indegree zero.

==
==
Session 2, 11:00--12:30
==
=======
Kruskal's algorithm

Constructing a minimum cost spanning tree. Sort the edges in
order of cost.

Pick up the edges in order of cost. The current partial spanning
tree is a collection of components. For each edge, if it creates
a cycle within a component, skip it. Otherwise, it either
creates a new component or connects two existing components.

How do we efficiently maintain the components and merge them?

==
=======
Disjoint-set data structure (Merge-find set)

Universe U = {1,2,...,N}

Initially, U is divided into N singleton subsets
{{1},{2},...,{N}}.

We want to maintain a collection of disjoint subsets of U
(components) to support the following operations.

a) Given an element, find which subset it belongs to (return a
   unique identifier for that subset)

b) Merge two subsets (union)

We could maintain an array Comp[1..N] so that Comp[i] is a unique
label identifying the subset that i belongs to. We can check
whether two vertices are in the same component in constant time.
However, to merge two components, we have to traverse Comp and
make the labels of the two components the same. This takes time
O(N), which is too expensive --- it will take MN steps across M
merges.

Alternatively, for each label (component), we maintain a (linked)
list of vertices that have that label. This will not help us
much because the size of each component could be about N.

Suppose, in addition, we maintain an array Size that records the
size of component. When we merge two components, we merge the
smaller component into the larger one --- that is, we change the
labels of the smaller component to be the same as the larger one.
(For this, we assume that we maintain, for each component, both
an array mapping vertices to components and a list of the
elements that belong to that component. We use the list to
efficiently identify all the vertices in the component to be
renamed and update each of them in the array in constant time, so
that "find" remains a constant time operation.)

The find operation continues to be constant time.

What is the cost of union?   Naively, each union could take N
time, as before. However, let us count more carefully, across
all the merges.

Each union relabels every element of the smaller set. How many

times does the label of an element change? Each time the label
of an element s changes, the size of the component that s belongs
to (at least) doubles. Thus, in log N steps, the component that
s belongs to will be the full set, so no element can be
relabelled more than log N times. Thus, over any sequence of
unions, the total number of relabel operations is at most N log
N.

==
=======
Path Compression

With each element, store a pointer to the component that it
belongs to. The last element in a component, points to itself.

Initially, each element is a singleton component and points to
itself.
```
   ----------  ----------  ----------       ----------
 | 1 | self | | 2 | self | | 3 | self | ... | N | self |
   ----------  ----------  ----------       ----------
```

When we merge, we make pointer for the head of one component
point to the head of the other one. For instance, if we have
components {1,2,3,4} and {5,9} as on the left, we can merge
component 5 into component 1 by making the pointer from 5 point
to 1, as shown on the right. (In the picture, all edges point
up.)


```
   Component 1    Component 5    Component 1 after merge

   {1,2,3,4}        {5,9}          {1,2,3,4,5,9}

     1 self        5 self            1 self
     / \             |              / | \
    3   4            9             3   4   5
    |                             |       |
    2                             2       9
```

Merge thus takes constant time.

What about find? To find the component of an element, we start
with the pointer attached to that element and walk to the root of
the tree to get the identity of the component to which it
belongs.

If we merge from N,N-1,...2,1 in that order

   ((... ((N U N-1) U N-2) ....) U 2) U 1)

we could end up with a tree consisting of a single path

   N --> N-1 --> N-2 --> N-3 --> ... --> 3 --> 2 --> 1 self

so find could take time O(N).

Once again, we can maintain the size of each component and insist
on merging smaller components into larger components. This will
guarantee that the height of a component is only (log N).

Alternatively, we can ignore the sizes of the components and do
"path compression". When we find an element for the first time,
we walk up a path to the root of the component to identify the
component. Now, we make a second pass from that element and
replace pointers along the way to point directly to the root.
This corresponds to "compressing" the path from each element
along the way to a direct edge to the root.

It turns out that with path compression, a sequence of K finds
will take time about 4K. (Strictly speaking, the "constant" 4 is
not a constant but a very very slow growing function of K.)

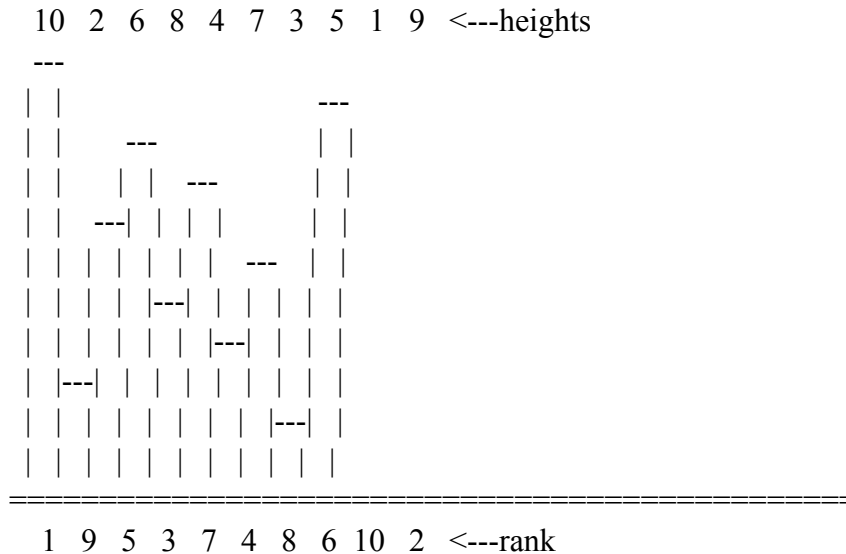In this representation, we don't need to maintain sizes --- it
works whichever direction we merge. Also, we don't need to
maintain a list of elements of each component.

This is called the "union-find" or "merge-find" data structure.
==
=======

Advertisement hoardings (IOI Training Camp 2006)

Given the heights of buildings along a road, find the horizontal
rectangle with largest area that fits within the building.

```
  10  2  6  8  4  7  3  5  1  9  <---heights
  ---
  | |                     ---
  | |        ---          | |
  | |       | |  ---      | |
  | |   ---| | | |        | |
  | | | | | | | |  ---    | |
  | | | | |---| | | | | | |
  | | | | | | |---| | | | |
  | |---| | | | | | | | | |
  | | | | | | | | | |---| |
  | | | | | | | | | | | |
  =================================================
   1  9  5  3  7  4  8  6 10  2  <---rank
```

Solution:

Sort the buildings by height. Inductively collect buildings into
components. Initially, buildings do not belong to any component.
Maintain for each component the size of the component.

Start with the tallest building and create a component for it.
For each building, check if its neighbours belong to an existing
component. If not, create a new component. If a neighbour is a
component, merge this building with the adjacent component. (If
both neigbhours are components, merge both neighbours and this
building into a single large component.)

Since the current building is shorter than any building in the
neighbouring component, the largest new rectangle that fits in
the extended component is height of this building times number of
elements of the extended component.

Maintain a global variable MaxRectangle that has the size of the
largest rectangle seen so far. Each time we add a building, we

update this value if the new rectangle is larger than the current maximum.

==
=======

Pyramid (IOI 2006)

After winning a great battle, King Jaguar wants to build a
pyramid that will serve both as a monument to remember his
victory and as a tomb for the brave soldiers that died in
battle. The pyramid will be built in the battlefield and will
have a rectangular base of a columns by b rows. Inside it, at
ground level, is a smaller, rectangular chamber of c columns by d
rows that will contain the corpses and weapons of the fallen
soldiers.

The King's architects have surveyed the battlefield as an m
columns by n rows grid and have measured the elevation of each
square as an integer.

Both the pyramid and the chamber are to be built covering
complete squares of the grid and with their sides parallel to
those of the battlefield. The elevation of the squares of the
internal chamber must remain unchanged but the remaining terrain
of the base of pyramid will be leveled by moving sand from higher
squares to lower ones. The final elevation of the base will be
the average elevation of all the squares of the base (excluding
those of the chamber). The architects are free to locate the
internal chamber anywhere within the pyramid as long as they
leave a wall at least one square thick surrounding the chamber.

Help the architects pick the best place to locate the pyramid and
the internal chamber so that the final elevation of the base is
the maximum possible for the sizes given.

The figure shows an example of an 8 x 5 battlefield with a x b =
5 x 3 and c x d = 2 x 1; the number in
each square represents the elevation of the terrain in that
particular position of the field. The larger marked rectangle the
base of the pyramid while the inner marked rectangle squares
represent the chamber. This figure illustrates an optimal

placement.

```
                  ------------------------
    1   5  10 |3   7   1   2     5|
        |       --------       |
    6  12   4 |4   3 |3   1|    5|
        |       --------       |
    2   4   3 |1   6   6  19     8|
        ------------------------
    1   1   1   3   4   2   4     5

    6   6   3   3   3   2   2     2
```

Given the dimensions of the field, the pyramid, and the chamber
along with the elevation of every square in the field, find the
location of both the pyramid in the field and the chamber inside
the pyramid so that the elevation of the base is the maximum
possible.

Limits: $3 <= m,n <= 1000$

Solution:

We can do some preprocessing and compute in MN time for each
(i,j) the size of the a x b rectangles and c x d rectangles with
top left corner at (i,j).

Now, for each a x b "base" we have to find the smallest c x d
"chamber" that we can place within this "base". Since we have to
leave a wall at the boundary, we effectively need to find the
minimum c x d rectangle within this subrectangle of size (a-2) x
(b-2) sitting inside the base. Henceforth, let us denote the
size (a-2) x (b-2) as e x f.

Let cd(i,j) denote the size of the c x d rectangle at position
(i,j). We want to find the minimum such value in each e x f
rectangle in the whole grid. We can do this efficiently in MN
time by tiling the whole space as e x f rectangles and computing
four quantities for each point inside each tile (see
two-dimensional version of min-segment, Session 2, Day 3).

So, overall the problem can be computed in MN time.

Yertle the Turtle (ACM)

Yertle, the turtle king, has 5607 turtles. He wants to stack up
these turtles to form a throne. Each turtle i has weight W[i]
and capacity C[i]. A turtle can carry on its back a stack of
turtles whose total weight is within C[i]. This should include
W[i], the weight of the current turtle, so assume that C[i] >=
W[i] always.

Find the height of the tallest turtle throne that Yertle can
build.

Solution:

Define residual capacity of turtle i as C[i] - W[i].

Not enough to look at heuristics such as "place min weight turtle
on top" or "put max residual capacity turtle on bottom".

For instance:

  W   C
 100 100   <-- Here min weight turtle comes below
  3   103

10^6 10^6+10^3 <-- Here max residual capacity should go on top
   1   10^4

The problem numbers suggest that we can at most use N^2 time.

Claim: If there is a stack of height k, there is a stack of
  height k in which carrying capacities are sorted in descending
  order from bottom to top.

Proof: Suppose, t1 is below t2 and t1 has smaller carrying
  capacity than t2. Then, we can swap t1 and t2 and the stack
  will still be stable.

We will return to this problem later.

==
=======

Paint (revisited)

A painter paints a strip N squares wide

```
         ----------------------
         | 1 | 2 | 3 | .... | N |
         ----------------------
```
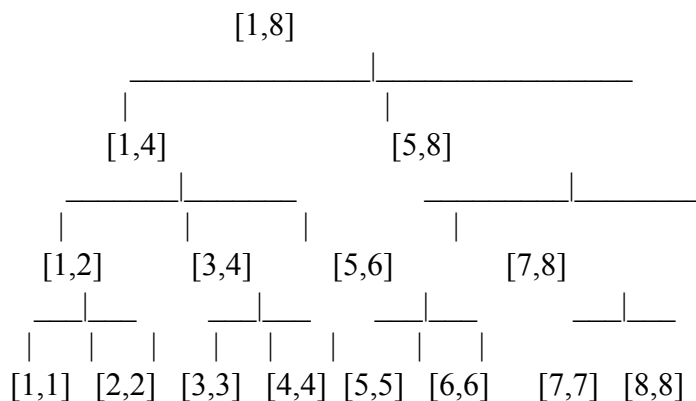
In one step, the painter paints a continuous strip from square i
to square j, which we denote [i,j]. Periodically, we want to
answer the following query:

  minlayers([i,j]) --- among all squares in the interval [i,j],
            report the minimum number of layers among
            the squares in this interval

How do we efficiently answer these queries?

Solution

Recall that we recorded the paint strokes in a segment tree.

```
                     [1,8]
          _____|_____
          |                      |
        [1,4]                  [5,8]
     _____|_____          _____|_____
     |          |          |           |
   [1,2]      [3,4]      [5,6]       [7,8]
   __|__      __|__      __|__       __|__
   |   |      |   |      |   |       |   |
 [1,1] [2,2] [3,3] [4,4] [5,5] [6,6]  [7,7] [8,8]
```

  At each vertex v, when we apply a layer of paint, we record

coats(v) = number of paint strokes that contain the entire
            interval Interval(v), but do not contain the entire
            interval Interval(parent(v)) spanned by the parent
            of v.

The number of coats at an interval is given by the value at
this node and the values of intervals that include this one
(these are above).

We need to store more information at each node to answer
minlayer queries. What is the structure of this information
and how do we use it to answer queries?

===
=======

Longest open segment

There are N plots arranged in a sequence 1..N. As time goes by,
these plots are sold, one at a time. Periodically, you have to
report the longest unsold segment of plots.

For instance, if N = 15 and you get the instructions

  sell(3)
  sell(8)
  sell(7)
  sell(15)
  report()

the answer is 6, which is the length of the empty interval
[9,14].

Solution

Construct a segment tree whose leaves correspond to the plots.
For each interval I, maintain:

  longest(I)   : length of longest open segment in I
  longestleft(I) : distance to first sold plot from left end of I

longestright(I) : distance to first sold plot from right end of I

The final answer we want is longest([1..N])

When we sell a plot, we have to update these four quantities.

```
 // longest(I) is either max longest from left or right, or spans the
 // two subintervals
 longest(I) = max(longest(left(I)),
            longest(right(I),
            longestright(left(I)) + longestleft(right(I)))


 // longestleft(I) is longestleft(left(I)) unless entire left interval
 // is empty, which is checked by asking if longestleft(left(I)) is
 // equal to length(left(I))
 longestleft(I) = if longestleft(left(I)) = length(left(I))
            then
              length(left(I)) + longestleft(right(I))
            else
              longestleft(left(I))


 // symmetric computation for longestright(I)
 longestright(I) = if longestright(right(I)) = length(right(I))
             then
               length(right(I)) + longestright(left(I))
             else
               longestright(right(I))
```

Exercise: What if we can demolish houses and restore a plot to
        being empty?

==
=======

Rivers (IOI 2005)

Nearly all of the Kingdom of Byteland is covered by forests and
rivers. Small rivers meet to form bigger rivers, which also meet
and, in the end, all the rivers flow together into one big
river. The big river meets the sea near Bytetown.

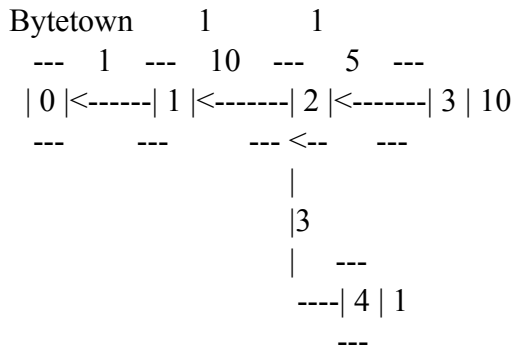There are N lumberjacks' villages in Byteland, each placed near a

river. Currently, there is a big sawmill in Bytetown that processes all trees cut in the Kingdom. The trees float from the villages down the rivers to the sawmill in Bytetown.

The king of Byteland decided to build K additional sawmills in villages to reduce the cost of transporting the trees downriver.

After building the sawmills, the trees need not float to Bytetown, but can be processed in the first sawmill they encounter downriver. Obviously, the trees cut near a village with a sawmill need not be transported by river. It should be noted that the rivers in Byteland do not fork. Therefore, for each village, there is a unique way downriver from the village to Bytetown.

The king's accountants calculated how many trees are cut by each village per year. You must decide where to build the sawmills to minimize the total cost of transporting the trees per year. River transportation costs one cent per kilometre, per tree.

Example:

```
Bytetown     1         1
   ---   1   ---   10   ---     5   ---
  | 0 |<------| 1 |<-------| 2 |<-------| 3 | 10
   ---        ---          --- <--       ---
                            |
                            |3
                            |   ---
                            ----| 4 | 1
                                ---
```

Nodes are villages, {1,2,3,4}. Edges are rivers. Number of trees to be cut is written by the side of each village. Length of each river is written by the edge. 2 sawmills are to be placed.

Solution in this case is to place the mills at villages 2 and 3, which incurs a cost of 4.

Input:
  The number of villages N (<= 100), the number of additional sawmills to be built K (<= 50), the number of trees cut near

each village W_i (<= 10000), and descriptions of the rivers,

Output:
  The minimal cost of river transportation after building
  additional sawmills,

Solution:

Observe that the rivers form a tree.

Use dynamic programming. What are the important parameters?

Need to consider every possible village as a candidate. If we
place a sawmill at v, it "controls" a subtree of which it is the root.
What is the cost of this subtree?

We need to know how many sawmills have already been placed in the
subtree rooted at v, say r.

Also need to take care of the next sawmill downstream between v
and Byteland, where the logs at v will land up if there is no
sawmill at v. Instead of considering the identity of the village
downstream, we will use as a parameter the distance L of that
village from Byteland. This uniquely specifies the village.

As a simplifying assumption, assume we have at most two children
at each node. We compute two quantities.

Thus, we want to compute two cost functions.

Cost1(v,r) == min cost in subtree rooted at v with r sawmills
          with one mill at v

Cost2(v,r,L) == min cost in subtree rooted at v with r sawmills
          without one mill at v, where L is the depth from
          Byteland of the nearest sawmill downstream from v

Let v have children w1 and w2.

Then
              i sawmills below w1   i sawmills below w1
               one sawmill at w1   no sawmill at w1

```
                    \            /
Cost1(v,r) =    min       { min[ Cost1(w1,i),Cost2(w1,i,depth(v))],
          i,j s.t i+j=r-1   min[ Cost1(w2,j),Cost2(w2,j,depth(v))] }
                   /          \
              j sawmills below w2    j sawmills below w2
              one sawmill at w2      no sawmill at w2
```

Then

```
Cost2(v,r.L) =  Trees(v)*(depth(v) - L)  /*  Cost of trees at v
                              travel upstream    */
          +

          min       { min[ Cost1(w1,i),Cost2(w1,i,L)],
        i,j s.t i+j=r    min[ Cost1(w2,j),Cost2(w2,j,L)] }
```

Now, what happens when we have more than 2 children, we have to analyze all ways of splitting r among the children.

How do we efficiently deal with the problem of considering all partitions of K as K_U?

Restructure the children of a node as a skew tree:

```
  -- v--                  v
  / /\ \      ===>       /\
 u1 u2 u3 u4           u1   x1
                      / \
                    u2   x2
                     / \
                    u3   x3
                     /
                    u4
```
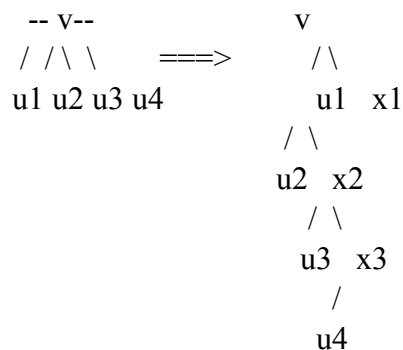
We have reduced the degree of each node. We set the number of trees cut at {x1,x2,x3} to be 0. We copy the costs of (u2,v), (u3,v), (u4,v) to the edges (u2,x1), (u3,x2), (u4,x3), respectively and set cost of the new edges (x3,x2), (x2,x1), (x1,v) to be 0. This ensures that the modified tree has the same costs as the original one.

In this way, we can transform the original tree into a new one with the same cost function for which the cost is easier to compute.

This process adds one node per child, so this at most doubles the number of villages.

What happens if we put a sawmill at a fictitious village xi? We can argue that we can shift this sawmill upto v and not change the complexity.

```
====================================================================
=======
====================================================================
=======
```

```
====================================================================
=======
```

Session 1, 09:00--10:30

```
====================================================================
=======
```

Critical Intersections

Network of roads, where the roads have been relaid but work still has to be done at the intersections. Relaying an intersection will prevent traffic from flowing through that intersection. An intersection is critical if blocking that intersection disconnects two points. Want to avoid relaying a critical intersection during the day time.

Example 1:

```
    a---------b
    |\    |
    | \   |   e------------g
    |  \  | / \      / \
    |   \|/   \   /   \
    c-------- d -------f   h-------i
```
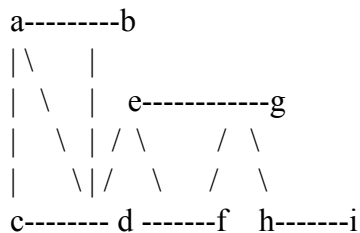
Here, {d,e,g} are critical intersections.

Example 2:

```
  a---------f--------g
  |     /|\    |
  |    / | \   |
  | /   |  \ |
  |/    |   \|
  b-------- d      h
  | \
  |  \
  c--- d
```
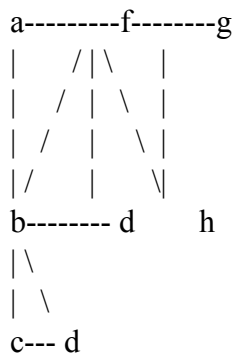
Here {b,f} are critical intersections.

How do we identify critical intersections?

Naive solution:

Delete each intersection and do a bfs/dfs to check if the graph remains connected. Take time $O(MN)$.

Critical intersections are vertices that disconnect the graph if they are deleted. These are called "articulation points" or "cut vertices".

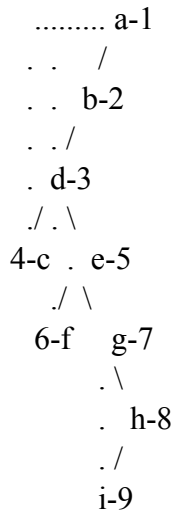Articulation points can be identified in a single dfs.

The root of the dfs tree is an articulation point iff it has more than one child. If there are two children, there cannot be an edge from one subtree to the other, because non-tree edges are back edges and can only go to an ancestor.

When is a non-root vertex v an articulation point? Can we generalize the argument for the root?

When there is a child w of v in the subtree such that the highest vertex that w can reach by going forward along DFS edges and then taking *one* back edge lies at or below v. In other words, if v is removed, w and all vertices below w become disconnected. An alternative way of saying this is that we want to look for a child w of v such that there are no back edges from the subtree rooted at w that go strictly above v.

Let us construct a dfs tree rooted at a for the examples above.
For each vertex, we have written its dfs number. Back edges are
shown by ... edges.

Example 1:

```
    ......... a-1
    . .    /
    . .  b-2
    . ./
    . d-3
    ./ . \
   4-c . e-5
     ./ \
    6-f  g-7
        . \
        .  h-8
        . /
        i-9
```

Here, d is an articulation point because it has a child e that
satisfies the criterion above. Similarly, e is an articuation
point (child g satisfies criterion) and so is g (child h
satisfies the criterion).

Example 2:

```
       b-1........
      / .. \    .
     / .. \   .
   c-2 . .    e-4 .
    | . .     | .
    |. .      | .
   d-3    .   f-5 ......
       . / \   .
       . / \   .
      a-6    g-7 .
              | .
              | .
            h-8 .
```
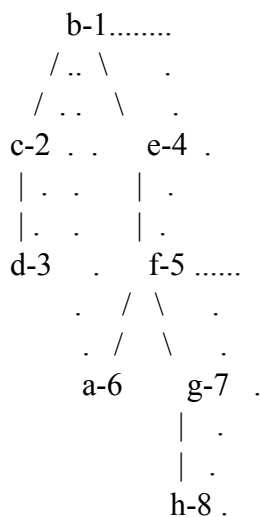
Here, b is an articulation point because it is the root and it

has more than one child. f is an articulation point because it
has a child g that satisfies this criterion.

For a node v, let low(v) denote the lowest indexed vertex
reachable from v, via a sequence of 0 or more tree edges followed
by a single back edge. This can be computed recursively as
follows:

low(v) =
    min({low(u) for all children u of v},
        smallest indexed node reachable directly a back edge from v)

To check if v is an articulation point:

  For each child u of v, check if low(u) >= DFS_number(v).

If there is at least one child for which this happens, then that
child's subtree will get disconnected if u is removed, so u is an
articulation point.

From this it follows that v is an articulation point if

    max   low(u)   >= DFS_number(v)
  u in child(v)

Let us call this quantity maxlow(v).

Computing low(u) and maxlow(u) can be built in to DFS.

```
  count = 1;

  DFS(u)
  {
   visited[u] = true;

   DFS_number[u] = count; count++;  /* Assign unique
                        DFS number to node */
   low[u] = u;

   for each edge (u,v) {

    if (visited[v]){
```

```
        if (v != parent[u]){              /* Take min with back  */
          low[u] = min(low[u],DFS_number(v));/* edges from u.  Note */
        }                                  /* that (u,parent(u))  */
                                           /* is not a back edge  */


      }else{

        parent[v] = u;                    /* Set parent */

        DFS(v);                           /* Recursively
                                          check children */
        /* At this point, if low[v] >= low[u], report that u is
           an articulation point */

        low[u] = min(low[u],low[v]);

        maxlow[u] = max(maxlow[u],low[v]);
       }
     }
    return;
   }
```
One way roads

Given a network of intersections and roads, turn all the roads
into one-way roads such that every pair of intersections is still
connected by a path. If it is possible, give such an
orientation, else print NO.

Solution:

Observations:

1. If there is a solution, every pair of vertices must be part of
   a cycle in the original undirected graph.

   (For every pair of vertices u and v, there is a path from u to
   v and a path from v to u. Let i-->j be a directed edge on the
   route from u to v. There must be a path that returns from v
   to u without using this edge.)

2. Can we use the DFS tree to orient the edges? Orient tree
   edges down and back edges up. This will work, if the original

graph can be oriented.

How do we check, however, if the graph can be oriented. If not, how do we identify the "bad" edges?

Consider an example.

```
       b
      /\
     /  \
    a    c
     \  /\
      \/  \
      d-----e
      /\   |
     /  \  |
    i-----j |
          f
         /\
        /  \
       g-----h
```

DFS tree:

```
   [1] a..........
        |       .
   [2] b        .
        |       .
   [3] c.......... .
        |        ..
   [4] e---------- .
        |      \.
   [5] f.   [8] d.
        |.      |.
   [6] g .  [9] i .
        |.      |.
   [7] h.   [10] j.
```
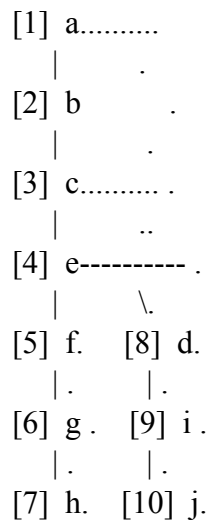
Here, if we orient tree edges down and back edges up, we cannot get from {f,g,h} to the rest of the graph.

This is because the edge (e,f) is a "cut-edge" or a bridge. The graph becomes disconnected if (e,f) is deleted.

Observe that low(f) = 5 is strictly greater than DFS_number(e) = 4. It is not difficult to argue that this is a general property of bridges.
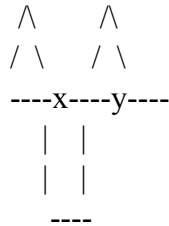
  A tree edge (u,v) is a bridge iff low(v) > DFS_number(u)

The end points of a bridge will always be cut vertices. However, it is not necessary that an edge joining two cut vertices is always a bridge. For instance:

```
      /\      /\
     / \    / \
    ----x----y----
       |  |
       |  |
        ----
```

Here x and y are cut vertices, but the edge (x,y) is not a bridge.

Example 2:
                        DFS

```
  a---------f--------g----i        a1.....
  |     /|     | /        |    .
  |    / |     |/      ..b2     .
  | /  |     h      . |\   .
  |/    |              .  c3  e5  .
  b-------- d                . |  | .
  |\                    ..d4  f6.
  | \                         |
   c--- d                    g7.
                              | .
                             h8 .
                              | .
                             i9.
```
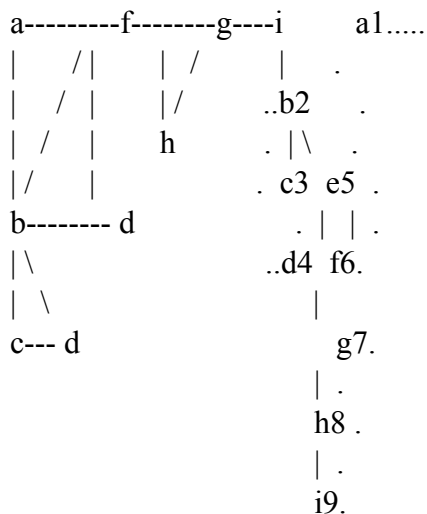
f-g is critical because low(g) < DFS_number(f)
==
===

Placing rooks

Given a chess board in which some squares are obstacles (#) and
other are holes (O). You have to place the maximum number of
rooks on this chess board such that no pair of rooks attack each
other.

1. A rook cannot be placed on a hole.

2. A rook attacks the row and column on which it is placed. The
   attacking range of a rook can span a hole, but is blocked by
   an obstacle.

Here is an example (not maximum!)

```
    -------------------------------
    | | | | | | | |R| |
    -------------------------------
    | |R| | | | | | | |
    -------------------------------
    |R|#|R| | | | | |
    -------------------------------
    | | | | | | | |O| |
    -------------------------------
    | | | | |#| | | |
    -------------------------------
    | | | | |R| | | |
    -------------------------------
    | | | | | | | | |
    -------------------------------
```

Solution:

Without obstacles and holes, the solution is N rooks, each on a
separate row and column.

Think of a graph where the vertices are the rows and columns of
the chessboard. There is an edge between a row and a column if
they intersect.

Since we have holes and obstacles, we modify this graph. The
vertices are now maximal vertical and horizontal segments between

obstacles. There is an edge connecting a vertical and horizontal segment if they intersect at a square that is not a hole.

Notice that this graph has a nice structure---we can separate the vertices into two disjoint sets, rows and columns. All edges go from one set to the other. There is no edge within a set. Such a graphs is called a bipartite graph --- a graph where we can partition the vertices as V0 and V1 and all edges go only from V0 to V1.

Each square on the chessboard corresponds to an edge in the graph. If we place a rook at a square, we select the corresponding edge. This rules out choosing any other edge that shares an endpoint with this edge. Let us say that two edges are disjoint if they have no common end points. To place all rooks, we have to choose a set edges that are disjoint.

A disjoint set of edges in a graph such that no two edges share a vertex is called a matching.

  Maximal matching: A matching that cannot be extended further
  Maximum matching: A matching whose size, in number of edges is
            maximized
  Perfect matching: all vertices are part of some edge in the set

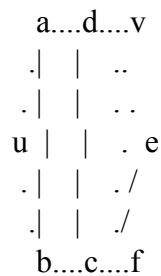So, the solution seek is a maximum matching in a bipartite graph.

===
=======
Finding a maximum matching in a bipartite graph
----

1. Start with an empty matching.

2. Assume we have selected some edges for our matching and we
   cannot find any disjoint edge to add to the current matching.

   Say that a vertex is matched if it is covered by the current
   matching. Otherwise, it is unmatched.

   For instance: u and v are unmatched below (the edges marked
   --- are in the current matching, the edges marked ... are not

in the matching).

```
       a....d....v
       .|   |  ..
       .|   |  . .
       u |   |  . e
       .|   |  ./
       .|   |  ./
        b....c....f
```

To extend this matching further we identify an alternating (augmenting) path.

An alternating (augmenting) path is a path in the graph that

-- starts and ends at unmatched vertices (i.e., they are not part of the current matching)

  Thus, the first and last edges in the path are not part of the current matching.

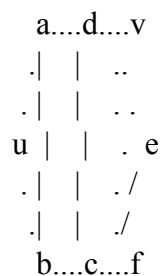-- the edges in the path alternate between matching edges and non-matching edges

An alternating (augmenting) path looks like the following.

```
..... ----- ..... ----- ..... ----- .....
```

    ..... = edges that do not belong to current matching
    ----- = edges that belong to current matching

For instance: u-a-b-c-d-v is an augmenting path

```
       a....d....v
       .|   |  ..
       .|   |  . .
       u |   |  . e
       .|   |  ./
       .|   |  ./
        b....c....f
```

If we find an alternating (augmenting) path, we invert the

sense of all edges in the path---matching edges become non
matching and vice versa. Thus,

     ..... _____ ..... _____ ..... _____ .....

becomes

      _____ ..... _____ ..... _____ ..... _____

and we increase the matching by one.

In the example above, we invert the path u-a-b-c-d-v to get

```
         a....d----v
         /.    ..
        /.    . .
       u  .   . e
        .. .  ./
         .. . ./
          b----c....f
```

We stop when no alternating (augmenting) paths can be found.

How do we find an alternating (augmenting) path?

==
=======
Session 2, 11:00--12:30
==
=======
A Multiplication Game (ACM competition)

This is a two player game. They start with the number X = 1 and
play alternately. A move consists of picking a number j in the
range {2,3,...,9} and updating X to X*j.

For instance, here is a possible evolution of the game:

```
            X
          -----
Start with   :   1
P1 plays 2   :   2
```

```
P2 plays 7   :  14
P1 plays 2   :  28
P2 plays 3   :  84
...
```

You are given a number N. The first player who makes X >= N
wins.

Given N, and assuming that both players play optimally, compute
which player will win.

Solution:

Whoever plays in the interval N/9 <= X < N wins, because he can
make X cross N in one move. Whoever plays in the interval N/18
<= X < N/9 loses, because he cannot reach N in one move but he
also cannot avoid pushing X into the winning range N/9 <= X < N.
In the same way, the interval N/(9*18) <= X < N/18 is winning,
because the player who plays in this interval can force the next
move to be in the losing interval N/18 <= X < N/9.

In this way, we can label the intervals back from N

```
     Win        Lose        Win
... N/162 <----> N/18 <----> N/9 <----> N
```

Now, look at whether 1 falls in a Win or Lose interval --- this
can be done in log N steps. Depending on this, P1 (the player
who moves first) wins or loses.

===
=======
Rectangle (IOI 2005, Poland)

Two people start with a slab of chocolates with M rows and N
columns and play the following game (with M,N <= 10^8). The
players take turns breaking the piece of chocolate. Each person
can cut the rectangle into two pieces, using either a single
vertical or single horizontal line. After each cut, the piece
with smaller area is discarded and the game continues with the
bigger piece. Eventually, the slab will reduce to a 1x1
rectangle. The player who has to play with a 1x1 rectangle loses

the game.

Develop a winning strategy.

Some positions are winning, some are losing.

Losing positions:

-- In one dimension, 1x3 is losing because we have to leave 1x2
   which will return to us as 1x1. Since 1x3 is losing,
   positions 1x4,..,1x6 are winning since we can leave the
   opponent with 1x3. The next losing position is 1x7, because
   we have to leave the opponent with 1x4,...,1x6 which are
   winning. Similarly, positions 1x7, 1x15, 1x31, ... are
   losing.

-- A square slab is always a losing position. However we cut,
   the opponent can reduce it back to a square. Eventually, the
   square will reduce to 1x1 which is losing.

-- A rectangle of size n x m, n+1 <= m <= 2n, is winning, since
   we can reduce it to a square of size n x n.

```
         m
     ------------
     |        |
   n |        |
     |        |
     ------------
```
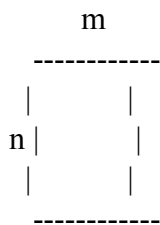
   Generalizing the 1-D case, n x (2n+1) loses, n x (4n+3) is
   losing etc. In general, n x m is losing if m = $2^k(n+1) - 1$.

   What if the rectangle is n x (2n+1) and the opponent reduces n
   to n'? Then, you can always reduce 2n+1 to 2n'+1 and leave a
   similar losing situation of n' x (2n'+1).

Losing position is a "trap" --- once a player enters a losing
position, no matter how he moves, the opponent

===
===

Let us look at games where we cannot exhaustively analyze the game in advance and write a closed form solution.

===
===
Bead Necklace (originally Pearls, CEOI 2003)

Blue dwarves and white dwarves, totally 1000 of them. The two types of dwarves are on opposite teams. One blue dwarf and one white dwarf found a bead necklace with red and black beads ending in a single pearl. Each team wants to get hold of the pearl.

To achieve this, they play a game:

Each dwarf D is given two lists, a red list R_D and a black list B_D. Both R_D and B_D are sets of dwarves (these sets may contain dwarves of either colour, but are guaranteed to be nonempty).

The necklace is given to one of the dwarves D. His moves are:

1. If the leftmost bead is the pearl, keep it and game ends.

2. If the leftmost bead is red, remove it and hand over the necklace (with leftmost bead removed) to any dwarf in the red list R_D.

3. If the leftmost bead is black, remove it and hand over the necklace (with leftmost bead removed) to any dwarf in the black list B_D.

You are given:

1. The sequence of beads in the necklace

2. The black and red lists of each dwarf

3. Who has the necklace at the beginning

Your aim is to determine whether the dwarf can ensure that the pearl eventually reaches another dwarf on his own team.

The current position of the game is completely described by the quantities (D,N) where

-- D: which dwarf has the necklace
-- N = b1 b2 .. bk: the sequence of beads in the necklace
             (without the pearl)

Call this a configuration. For each pair (D,N), label the configuration as W (winning) or L (losing). If the label is W, D can ensure that the pearl reaches his team. If the label is L, no matter what D does, the pearl reaches the other team.

We work backwards from the last bead in the necklace. If the necklace is currently a single pearl, the configuration is winning for all D.

Now, we step back once so that we have a configuration (D,bk). Depending on the identity of D and the lists he has, we know whether this is winning for the team that D belongs to.

-- If D is white, bk is red and there is at least one white dwarf
   D' in the list R_D, then D can revmove bk and hand the
   necklace to D', who gets the pearl. So, this configuration
   is winning.

   On the other hand, if R_D has no white dwarves, D must hand
   over the necklace with just the pearl to some blue dwarf,
   so (D,bk) is labelled L.

   Similarly, if bk is black, (D,bk) is winning if there is at
   least one white dwarf D' in the list B_D and (D,bk) is losing
   if there is no white dwarf D' in the list B_D.

-- Symmetrically, we can identify when (D,bk) is winning for an
   blue dwarf D.

In general, assume that we have calculated the status of (D,N) for N = b1 b2 ... bk. We can now calculate the status if we add an additional bead b0 to the left of N.

Suppose b0 is red and D is a blue dwarf. Then we can label (D,b0...bN) as W if either of the following hold:

-- there is a blue dwarf D' on D's red list such that
     (D',b1..bN) is labelled W. (Then we can k keep the bead in
     the same team and win in next configuration.)

  -- there is a white dwarf D' on D's red list such that
     (D',b1..bN) is labelled L. (Then we hand over the bead to
     the other team which is guaranteed to lose in next
     configuration.)

Otherwise, label (D,b0...bN) as L (D cannot avoid making a move
that leads to a losing position).

Make similar updates for the other three possibilities for b0 and
D --- (b0 is black and D is white), (b0 is black and D is blue)
and (b0 is red and D is white).

In this way, we can work backwards and label the initial
configurations at W or L.

===
===

Solving finite games

The bottom up computation in the Pearls game corresponds to the
following top down analysis, which applies for any game that has
a finite number of positions.

There are two players 0 and 1. The game is described as a game
graph. The vertices are the positions and each position of the
game is labelled 0 or 1 depending on which player moves at that
position. The edges describe how the game can evolve from one
position to another.

We label some of the positions as winning for player 0 --- let
this set be G. If the game ever reached G, player 0 wins.
Otherwise, player 1 wins. (If the game graph is a dag, all plays
will end in a sink state, so we just label the sink states as
winning for 0/1, as in the Pearls game.)

Let W0 denote the set of positions from which player 0 can win in
at most n moves. W0 is G, since, if the game is in a position in

G, player 0 has already won.

We now calculate W1.

- Consider any state s labelled 0 from which there is at least
  outgoing edge to t in W. Then, 0 can move into W = W0 and win, so s
  belongs to W1.

- Symmetrically, consider any state s labelled 1 from which all
  outgoing edges lead to W. Then, 1 is forced to move into W =
  W0 where 0 wins, so t belongs to W1.

In general, let Wi be the set of positions from where player 0 is
guaranteed to win in i moves. We can calculate W{i+1}.

- Consider any state s labelled 0 from which there is at least
  outgoing edge to t in Wi. Then, 0 can move into Wi and win in
  i moves, so s belongs to W{i+1}.

- Symmetrically, consider any state s labelled 1 from which all
  outgoing edges lead to some Wj, for j <= i. Then, 1 is forced
  to move into Wj from where 0 wins in j moves, so t belongs to
  W{i+1}.

Notice that W0 is included in W1 is included in W2 ... Since
the game graph has only n positions, for some i, we must have Wi
= W{i+1}. In particular, we must stop when we reach
W{n-1} because we cannot add more than n-1 new nodes to the
original set W0.

Let W the final winning set for 0 --- that is, W = Wi such that
Wi = W{i+1}.

Claim:
  a. If the game starts in W, player 0 always wins.
  b. If the game starts outside W, player 1 always wins.

Proof:
  a. If the game starts in W, player 1 cannot move the game
     outside W. Player 0 can not only stay in W but also
     "decrease the rank" --- that is, move from Wi to W{i-1}.
     So, after some time, the game must reach W0 where player 0
     wins.

b. On the other hand, if the game starts outside W, player 0
   cannot move the game to within W and player 1 can always
   keep the game outside W. Since player 0 can never reach G,
   player 1 must win.

How do we compute these quantities?

Naively, this can be done in O(nm) time --- we have to compute
sets W1, ... X{n-1} and we scan all edges in each iteration.

However, with some care, we can do it in O(m) time.

Hint: Keep a count at each neighbour of how many good neighbours
   it has. Initialize this count to 0. When you add a node to W,
   notify all your incoming neighbours that you are good
   (increment their good neighbour count).

   To classify a 0 vertex as good, check if its good count is nonzero.
   To classify a 1 vertex as good, check if its good count is
     equal to its degree.

   This can be done in linear time, like BFS.

===
===

Scores (IOI 2001)

We are given a directed graph, with start vertex is 1. Each
vertex is owned by Player 1 or Player 2. Each vertex has a value
(integer).

The game starts at 1. The player who owns vertex 1 starts the
game. At each vertex, the owner picks up the value on the vertex
and moves token to neighbouring vertex.

The game ends when the token returns to vertex 1. The winner is
the one who has picked up the maximum value during the game.

Condition: The graph is such that the game always terminates in a

finite number of moves.

Problem: Design an optimal strategy for Player 1 (who may or may not own Vertex 1)

Example

```
                    --
         Player 1 nodes are labelled |  |,
                    --
                    ==
         Player 2 nodes are labelled ||  ||
                    ==

         Values are stored inside the nodes.

         Vertex 1 is the one on the top left corner.


           ---        ===
         --| 1 |<-------|| 2 ||<-
         |  ---     -> ===   |
         |       _/        |
         |      /          |
         |    _/
         | --- /      ===    |
         ->| 3 |------->|| 4 ||--
           ---        ===
```
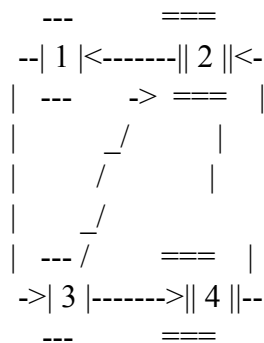
Solution

Since the game always ends by returning to vertex 1,there is no cycle that does not involve vertex 1. Otherwise, the players can stay within a cycle forever without the game terminating. Therefore, if we eliminate all edges coming into vertex 1, we have a dag.

Now, how do we evaluate positions bottom up.

Label each node with a pair of values (m1,m2) --- the interpretation is that if the game starts at this node, the maximum value that Player 1 can pick up is m1 and the maximum

value that Player 2 can pick up is m2. This node is winning for
Player 1 if m1 > m2.

We can initially evaluate these values for the leaf nodes in the
dag---those from which the only outgoing edge is back to vertex
1.

How do we propagate these values?

An internal node in the dag has successors in the dag for which
the game values are known. The node can also have a single back
edge to vertex 1.

Computing the (m1,m2) value for a node that belongs to Player 1:

Case 1 (There is no back edge from this node to vertex 1)

  Pick the successor in which Player 1 wins with his value
  maximum (i.e. in the successor node's label, Player 1's value
  is larger than Player 2's).

    Let this successor node be labelled (k1,k2). Let the value
    attached to the current node be l1.

    If l1 > k1 then Player 1 can afford to a play to a winning
    successor node labelled (n1,n2) in which n2 is minimum but n1
    is not necessarily maximum among all winning successors. In
    this case, we label the current node (l1,n2).

    Otherwise (l1 <= k1), Player 1 should move to the successor
    labelled (k1,k2), so we label the current node (k1,k2).

  If all successors of this node are losing for Player 1, pick
    losing successor (k1,k2) such that k2 is minimum. Let l1 be
    value of the current mode. Label this node with
    (max(l1,k1),k2).

Case 2 (There is a back edge from this node to vertex 1)

  As before, let (k1,k2) be the label of the best successor and
  let l1 be the value on the current node.

If l1 >= k1, label the current node (l1,0) [Player 1 will move
the game immediately back to Vertex 1 and win.]

If l1 < k1, label the current node (k1,k2) [Player 1 will
continue the game without returning to Vertex 1 at this
point.]

We also have to propagate labels to vertices that belong to
Player 2, using symmetric arguments.

===
==

===
=======
           IOI Training Camp, Lecture Notes, Day 7

                   Sat 28 Jun 2008
===
=======
Session 1, 09:00--10:30
===
=======
Depot (IOI 2001)

A storage depot consists of a 2 dimensional array of slots. Each
object to be stored in the depot has a unique serial number. When an
object with sequence number K arrives, it is stored according to the
following pair of rules.

1. If K is bigger than the sequence numbers in row 1, add K at the end
   of row 1.

2. Otherwise, find the smallest sequence number bigger than K in row
   1, say L. Replace object L by K and insert L into row 2 using the
   same pair of rules.

Problem: Given the final configuration of a depot, construct all
input sequences that could have given rise to the depot.

Solution:

Assume that the values in the depot are 1..N. In general, the number
of compatible input configurations is exponential in N.

A naive solution would be to enumerate all permutations of 1..N, apply
the depot rules and see if the resulting depot matches the one given.
This is possible within the time limits for N upto 7, possibly 8.

For larger inputs (upto 13), use a recursive procedure to work
backwards from the final depot. Notice that at each stage of
constructing a depot, the last element to move settles down at
the end of a row. It should also be a "corner" element---that
is, the row below should be strictly shorter in length.

Thus, for a given depot, for each row we look at the last element
and, if it is a corner, "undo" the sequence of moves that brought
it there, thus generating a possible last element of the input
sequence together with the previous configuration of the depot.
We recursively explore all these previous configurations. When
the depot becomes empty, we would have built up a possible input
sequence.

Example:

Input sequence : 3 2 6 4 1 5

Step :   1   2   3   4   5   6

Residual
Input: 2 6 4 1 5   6 4 1 5   4 1 5   1 5   5   Empty

Depot: 3   2   2 6   2 4   1 4   1 4 5
                 3   3   3 6   2 6   2 6
                                            3

Corner elements are {3,6,5}. Suppose we undo the 3. The 3
displaces the 2 in the previous row (largest element in that row
smaller than 3), which in turn displaces the 1 in the first row.
This leaves us with

   2 4 5   Partial input sequence: 1
   3 6

Corner elements are {6,5}. If we undo the 6, we get

2 4 6   Partial input sequence: 5 1
3

Corner elements are {3,6}. Undoing 6 again, we get

2 4   Partial input sequence: 6 5 1
3

Corner elements are {3,4}. Undoing 3, we get

3 4   Partial input sequence: 2 6 5 1

Now we have to undo 4, then 3, yielding a candidate input
sequence 3 4 2 6 5 1. Verify that this does indeed generate the
same depot.

Input sequence : 3 4 2 6 5 1

Resulting depot:

Step : 1   2   3   4   5   6

Residual
Input: 4 2 6 5 1   2 6 5 1   6 5 1   5 1   1   Empty

Depot: 3   3 4   2 4   2 4 6 2 4 5   1 4 5
                       3   3   3 6   2 6
                                               3

Generating Permutations

In many problems, we have to systematically search through a set of
possible "states" or "configurations" in order to find the answer.
The overall solution is a loop of the form

    while (there is one more state to examine){
      generate the next state;
      if (the new state has the desired property){
        do something useful;
      }

}

In many such search problems, one needs to be able to systematically generate all possible permutations of a given sequence of symbols.

For instance, given the sequence s = "a b c d", the set of permutations of s is {"a b c d", "a b d c", "a d b c", "a d c b", ..., "d c b a"}. Clearly, a sequence of length n generates n! permutations (we assume that all symbols in the sequence are distinct).

Our problem is to systematically generate all permutations of the sequence in lexicographic order; that is, the order in which the permutations would be listed in a dictionary. To achieve this, we would like a simple rule to determine the next permutation from the current one. We can then begin with the smallest permutation (arrange the symbols in ascending order) and keep applying this rule till we reach the largest permutation (all symbols in descending order).

Consider the following permutation over the letters {a,b,...,l}

       f g j l b e c k i h d a

How do we get the next permutation?

Begin from the right and keep moving left as long as the letters are increasing in value (where a < b < c < ... < l). In this case, start with a and scan past d, h, i, k. Stop at c because c < k.

Find the smallest value bigger than c in the sequence "k i h d a" after c and swap it with c. In this case, this value is d, so rewrite the last 6 letters of the sequence as "d k i h c a". Having done this, reverse the letters beyond d to get "d a c h i k".

The new permutation that we want is thus

       f g j l b e d a c h i k

Why does this work? Given values a1 < a2 < ... < ak, the smallest permutation is a1 a2 ... ak and the largest permutation is ak ... a2 a1.

For the element we have identified in the current permutation,

there is no way to increment the permutation keeping this element
fixed because we have already identified the largest permutation
to its right. Thus, we have to increment the current element
minimally, which is achieved by picking the next largest element
to its right.

How do we implement this?

1. Initially, sort the elements to obtain the minimum
   permutation.

2. Scan from the right to find which element to increment.

3. We now scan back to the right to find the next bigger element
   to replace this one. Observe that the sequence to the right
   is sorted in descending order, so this can be done using
   binary search.

4. Finally, we have to reorder the elements to the right in
   ascending order. If at Step 3 we swap the new value with the
   value we want to increment, the sequence to the right remains
   in descending order. We just have to invert this sequence to
   put it in ascending order --- no sorting is required.

An aside:

Suppose, while constructing a depot, we also keep track of the
order in which the squares are filled, which yields a second
"depot" with the same shape.

Input sequence : 3 4 2 6 5 1

Resulting depot:

Step : 1   2   3   4   5   6

Residual
Input: 4 2 6 5 1   2 6 5 1   6 5 1   5 1   1   Empty

Depot1: 3   3 4   2 4   2 4 6   2 4 5   1 4 5
                     3   3   3 6   2 6
                                        3

Depot2:   1   1 2   1 2   1 2 4   1 2 4   1 2 4
                             3   3   3 5   3 5
                                                      6

Since the second depot tells us the order in which the depot was
filled, we know in which order we should undo the moves, which
deterministically gets us the input sequence that generated this
pair of depots.

It turns out that each permutation generates a unique pair of
depots and vice versa, so the set of permutations of 1..n is in
1-1 correspondence with the set of pairs of valid depots of the
same shape over 1..n.

===
=======

Beads (APIO 2008)

Professor X has recently revealed his latest invention to the
world: the Ultimate Bead Swapper (UBS). As the name implies, it
can make a sequence of beads much more interesting by swapping
some beads in it! The UBS has N conveyor belts placed in
north-south direction in parallel. The conveyor belts are
numbered 1 to N from left to right. Every belt moves from north
to south at the same speed. There are M swappers placed between
adjacent conveyors. No two swappers are equally far from the
north end of the UBS. (In other words, they can be totally
ordered according to how far they are from the north end.) The
swappers are numbered 1 to M from north to south. The figure shows
the UBS when viewed from above.

Belt1 Belt2 Belt3 Belt4 Belt5
  | | | | |
  | <-Swap1-> | |
  | | | <-Swap2->
  | | | | |
 <-Swap3-> | | |

```
|    |      <-Swap4->      |
|    |    |    |    |
|    |      <-Swap5->      |
|    |    |    |    |
```
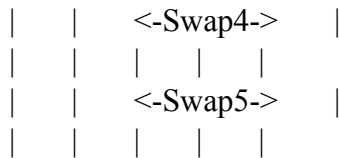
To use the UBS, N beads are placed at the north end of the
conveyor belts at the same time so that they form a horizontal
row as they move along the belt. When two beads come under a
swapper, the bead on the right conveyor belt is moved to the left
conveyor belt, and the bead on the left conveyor belt is moved to
the right conveyor. After being swapped, the two beads do not
break the horizontal row.

Task

Write a program that, given the number of conveyor belts N , the
number of swappers M , and the positions of each swapper, answer
Q questions of the form:

  Given K and J , for the bead that is placed on Belt K at the
  north end of the UBS, which belt is the bead on after all beads
  just move past Swapper J ?

Limits: 1 <= N, M, Q <= 300,000

Solution:

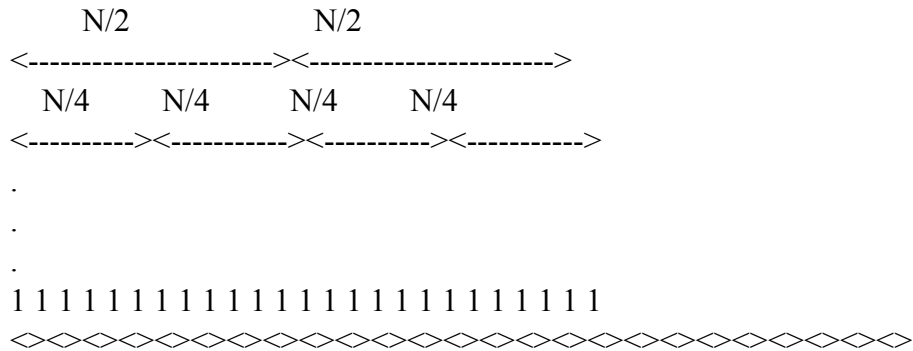The naive solution is to simulate (execute) the first J swaps for
each query. This takes time O(NQ) which is not acceptable.

First option:

A sequence of swaps defines a permutation. Store the
permutations for selected sequences. For each swapper i,
maintain the permutation corresponding to intervals of 1, 2, 4,
... , 2^k starting at i.

Even better, break up 1..N into blocks of size 1,2,4,8... and
maintain the permutations for these blocks.

                    N
    <--->
```

```
        N/2                    N/2
   <----------------------><---------------------->
      N/4        N/4        N/4        N/4
    <----------><-----------><----------><----------->
    .
    .
    .
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    <><><><><><><><><><><><><><><><><><><><><><><><><><>
```

Now, given J, we can tile the interval 1..J using log J such
blocks and compose the corresponding functions to get the overall
permutation. This takes O (Q log N) time to answer Q queries.

However, there are 2M such blocks. If we naively maintain a full
permutation per block it takes O(MN) space which is too much.

Instead, for each permutation we only keep track of the elements
that move in a sorted list. Having got a permutation, it now
takes log N time to determine where i moves in this permutation.
The good news is that the space requirement overall reduces to M
log M.

Second option:

We maintain a table with rows of the form

    Bead   Move   New belt

Each row in this table says "At step j, bead i is swapped and
moves to belt k". Each swap moves 2 elements, so across N swaps
we have only 2N moves. Therefore, this table has only 2N
entries. If we maintain the moves in sorted order, we can then
answer a query of the form "Where is bead i after step j?" by
doing a binary search among the moves of bead i in the table.

To maintain this table, use vectors.

===
=======

Mobiles (APIO 2007)

A mobile is a multi-layered decoration that is hung from the
roof. Each mobile consists of a series of horizontal rods
connected by vertical wires. Each rod has a wire hanging from
both ends, which holds either another horizontal rod or a toy.

A sample mobile is shown below:

```
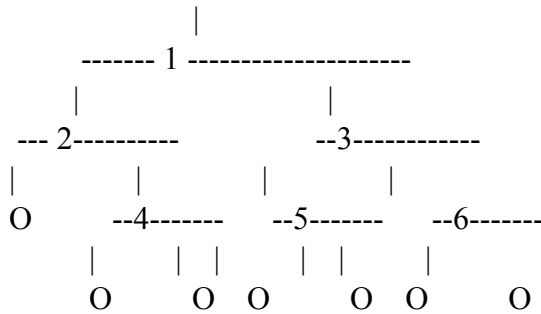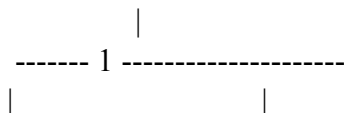                   |
          ------- 1 ---------------------
          |                   |
      --- 2----------         --3------------
      |         |         |         |
      O        --4-------   --5-------   --6-------
               |     | |     | |     |
               O     O O     O O       O
```

You need to find a mobile that has the following configuration.

(i) Any two toys are at the same level (i.e., are joined to the
    roof by the same number of rods), or differ by only one
    level;

(ii) For any two toys that differ by one level, the toy to the
    left is further down than the toy to the right.

Mobiles can be reconfigured as follows.

  The ends of any individual rod can be 'swapped' by unhooking
  whatever is hanging beneath the left and right ends of the rod,
  and reattaching them again at opposite ends. This process does
  not modify the ordering of any rods or toys further down.

The mobile above satisfies condition (i) but not condition (ii)
because the toy at the leftmost end is at a higher level than the
toys to its right. This mobile can be reconfigured to meet
condition (ii) using two reconfigurations, first swapping the
ends of rod 1 and then swapping the ends of rod 2.

```
                   |
          ------- 1 ---------------------
          |                   |
```

```
   --3------------          --- 2----------
    |         |          |         |
  --5-------     --6-------     --4-------        O
  |     | |     | |          |
  O     O O     O O        O
```

The task is to identify if a given mobile can be reconfigured to meet requirements (i) and (ii) and, if so, compute the minimum number of reconfigurations required to do so.

Solution

We can think of the mobile as a binary tree. A binary tree that meets requirements (i) and (ii) will have levels 1,2,..,k-1 filled completely and level k filled to some extent from left to right. Let us call such a tree a "mobile tree".

In a mobile tree of depth k, one of the two following cases must hold, depending on how many leaves are there are depth k.

(a) The left subtree of the root is a complete binary tree of depth k and the right subtree is a "mobile tree" of depth k-1.

(b) The left subtree of the root is a "mobile tree" of depth k the right subtree is a complete binary tree of depth k-1.

We can then inductively define the following quantities:

C(v): Minimum number of moves to make the tree rooted at v complete:

   This is 0 if v has a complete tree below it, and infinity otherwise.

M(v): Minimum number of moves to make the tree rooted at v a mobile tree.

   To determine M(v), we have to consider three cases.

   Case 1: depth(leftchild(v)) < depth(rightchild(v))

Need to swap left and right. After swapping, get a
complete tree on right and mobile tree on left

Case 2: depth(leftchild(v)) > depth(rightchild(v))

No swap needed, get a complete tree on right and mobile
tree on left.

Case 3: depth(leftchild(v)) = depth(rightchild(v))

May or may not swap. After swapping (if necessary),
left must be complete, right must be mobile tree.

This gives us the following update rule for M(v):

M(v) = 1 + C(leftchild(v)) + M(rightchild(v)), if Case 1
    M(leftchild(v)) + C(rightchild(v)),   if Case 2
    min {
      1 + M(leftchild(v)) + C(rightchild(v)),
      C(leftchild(v)) + M(rightchild(v)),
    },   if Case 3

==
=======

Backup (APIO 2007)

You need to back up computer data for offices situated along
single street. You decide to pair off the offices. For each pair
of offices you run a network cable between the two buildings so
that they can back up each others' data.

However, your local telecommunications company will only give you
K network cables, which means you can only arrange backups for K
pairs of offices (2K offices in total). No office may belong to
more than one pair (that is, these 2K offices must all be
different). Furthermore, the telecommunications company charges
by the kilometre. This means that you need to choose these K
pairs of offices so that you use as little cable as possible.

As an example, suppose you had five offices on a street, situated distance 1, 3, 4, 6 and 12 from the beginning of the street, and the telecommunications company will provides you with K = 2 cables. The best pairing in this example is created by linking the first and second offices together, and linking the third and fourth offices together. This uses K = 2 cables as required, where the first cable has length 3 − 1 = 2, and the second cable has length 6 − 4 = 2. This pairing requires a total of 4 of network cables, which is the smallest total possible.

Given the locations of N offices and the number of cables K that are made available, compute the cost in cable length of the best K way pairing. (N <= 100,000).

Solution

Observe that in the optimal solution any cable must run from an office to its neighbour. If a cable spans an office (or another connection), we can reconnect it in a shorter way.

```
   _____
  |       |
  w   x   y          =>  w---x   y



   _____
  |             |
  w    x-----y    z     =>  w---x    y-----z
```

This provides us an immediate dynamic programming solution.

Let Cable(j,m) be the optimal way of connecting offices 1..j using m cables.

Then

```
   Cable(j+1,m) = min {
            Cable(j,m),   // Do not connect office j
            Cable(j-1,m-1) + (dist(j) - dist(j-1))}
                  // Connect j to j-1
```

This yield an N*K algorithm, which is good enough for 60% of the

test cases, but not the full range.

Another approach is to greedily assign pairs according to shortest distance. This is wrong, but from the wrong solution, we can extract a better solution than the dynamic programming above.

Suppose the offices are arranged as follows

A <-11-> B <-5-> C <-3-> D <-6->E <-14-> F <-10-> G <-9-> H <-10-> I

and we have K = 4 cables.

A greedy solution would first pick CD = 3. This rules out BC and DE. The next best is GH = 9. This rules out FG = 10 and HI = 10, so we have to finish with AB = 11 and EF = 14 for a total length of 3+9+11+14=37.

On the other hand, we could have picked {BC=5, DE=6, FG=10, HI=10} for a total of 31.

Notice that if the minimum edge (CD, in this case) is not in the optimal solution, both its adjacent edges must be in the optimal solution (BC and DE, in this case). Suppose this is not the case --- that is, CD is minimum, but one of BC or DE is not used in the optimum solution. Then, either C or D is still free and we can replace BC by CD or DE by CD, depending on which one is available, and get a shorter solution.

This yields the following approach. We greedily add the edge CD = 3. Then we compensate for a possible mistake by adding a "virtual" edge "BE". The edge "BE" corresponds to undoing CD and adding instead the pair {BC,DE}. The cost of BC+CD = 11. The cost of CD is 3. If we say that adding CD and then adding "BE" is the same as BC+CD, we need the cost of CD+"DE" to be 11, so the cost of "BE" is 11-3 = 8.

We now have a partial greedy solution {CD} and a new problem involving A,B,E,F,G,H,I that looks like this.

A <-11-> B   C **3** D   E <-14-> F <-10-> G <-9-> H <-10-> I
                   |_____|

In this problem, the virtual edge "BE" is shortest, so we add
this to our greedy solution and get a new virtual edge "AF" of
length 11+14-8 = 17.

Soln = {CD,"BE"}

```
A <-11-> B    C **3** D    E <-14-> F <-10-> G <-9-> H <-10-> I
|     *            *         |
|     ******** 8 *******         |
  ----------------17--------------------
```

Using greedy, the next best edge is GH, so we add it and put in a
virtual edge "FI" of length 10+10-9 = 11.

Soln = {CD,"BE",GH}

```
A <-11-> B    C **3** D    E <-14-> F <-10-> G **9** H <-10-> I
|     *            *         |                 |
|     ******** 8 *******         |               |
  ----------------17-------------------- -----------11------------
```

For the fourth edge, we choose the smaller of the two virtual
edges "FI", and we have a solution for K = 4 as follows:

Soln = {CD,"BE",GH,"FI"}

This corresponds to doing CD, undoing it and replacing it by
{BC,DE}, doing GH, undoing it and replacing it by {FG,HI}, for a
final solution of {BC,DE,FG,HI}, which matches the solution we
pointed out earlier.

Why does this work? The idea is an extension of what we argued
about the overall minimum edge and its two neighbours. For each
edge we add, we preserve the possibility of undoing it in the
virtual edge, using the fact that undoing a greedy choice
requires adding both neighbours to the solution.

How do we implement this?

1. Maintain the current cost of edges in a heap. Also maintain
   a boolean array Used[1..N] where Used[i] is true if i has been

used up already.

2. At each step, delete the minimum cost edge (i,j) from the heap.

   If either i or j has already been used, we discard this edge
   and pick the next minimum edge from the heap.

   If both i and j are unused, we add (i,j) to our solution, mark
   Used[i] = Used[j] = True and add a virtual edge (i-1,j+1) with
   the appropriate cost to the heap.

Thus, we spend O(N) time building the initial heap and make K
updates to the heap, so the overall time is N + K log N.

===
=======

Coffee Shop (APIO shortlist 2007)

Given some stations arranged in a tree-like network, we want to
place coffee shops at some stations so that no station is more
than K hops away from a coffee shop.

Solution

Since the network is a tree, we identify any one node as the root
and walk up from the leaves.

When we come to a node v, we assume we have processed its
subtrees and placed coffee shops. We need to decide whether to
place a coffee shop at v.

The only reason to place a coffee shop at v is if there is some
node below v that cannot be serviced below v and would become
more than K hops away from a coffee shop if there is no shop at
v.

To compute this, we maintain two quantities:

  C(v) : minimum depth of coffee shop below v
  U(v) : maximum depth of unserved node below v

We first compute

$$C'(v) = 1 + \min_{w \text{ in } child(v)} C(w)$$

to find the minimum depth of a coffee shop from v without placing a coffee shop at v.

Let {w1,w2,...,wn} be the children of v. Even if we don't place a coffee shop at v, an unserved node below one child wi of v may get served (via v) by a coffee shop below some other child wj of v. Recall that we have already computed C'(v), the shortest distance from v to a coffee shop below it. For each child w of v, compute

$U'(w) = 0$,　if $U(w) + 1 + C'(v) <= k$ // node below w can reach
　　　　　　　　　　　// a coffee shop via v
$U'(w) = U(w)$,　otherwise　　　// cannot reach a
　　　　　　　　　　　// a coffee shop via v

Now, we must place a coffee shop at v if $U'(w) = k-1$ (otherwise, after adding v without a coffee shop, the unserviced node below w will be k hops away from v without seeing a coffee shop and nearest coffee). If we add a coffee shop at v, we have

$$C(v) = U(v) = 0$$

otherwise

$$C(v) = C'(v)$$
$$U(v) = \max_{w \text{ child of } v} U'(v)$$

Finally, when we reach the root, we must put a coffee shop if there is any unserviced node below it, or if it is itself unserviced by anything below it.

To show that this solution is optimal, consider any other solution. Inductively, in every subtree, if our solution puts a coffee shop, the other solution must have a matching coffee shop (because we put coffee shops only when absolutely necessary). Our coffee shops are pushed up as far as possible, which can only improve the total number.

Finding a maximum matching in a bipartite graph

A bipartite graph is one in which we can separate the vertices
into two disjoint sets, V0 and V1 so that all edges go only from
V0 to V1---there is no edge within either set.

Finding a matching:

1. Start with an empty matching.

2. Assume we have selected some edges for our matching and we
   cannot find any disjoint edge to add to the current matching.

   Say that a vertex is matched if it is covered by the current
   matching. Otherwise, it is unmatched.

   For instance: u and v are unmatched below (the edges marked
   --- are in the current matching, the edges marked ... are not
   in the matching).

```
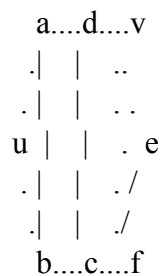        a....d....v
       .|   |   ..
       .|   |   ..
      u |   |   . e
       .|   |  ./
       .|   |  ./
        b....c....f
```

   To extend this matching further we identify an alternating
   (augmenting) path.

   An alternating (augmenting) path is a path in the graph that

   -- starts and ends at unmatched vertices (i.e., they are not
      part of the current matching)

      Thus, the first and last edges in the path are not part of

the current matching.

-- the edges in the path alternate between matching edges and non-matching edges

An alternating (augmenting) path looks like the following.

```
..... ----- ..... ----- ..... ----- .....
```

> ..... = edges that do not belong to current matching
> ----- = edges that belong to current matching

For instance: u-a-b-c-d-v is an augmenting path

```
    a....d....v
    .|   |  ..
    .|   |  ..
    u|   |  . e
    .|   |  ./
    .|   |  ./
     b....c....f
```

If we find an alternating (augmenting) path, we invert the sense of all edges in the path---matching edges become non matching and vice versa. Thus,

```
..... _____ ..... _____ ..... _____ .....
```

becomes

```
_____ ..... _____ ..... _____ ..... _____
```

and we increase the matching by one.

In the example above, we invert the path u-a-b-c-d-v to get

```
     a....d----v
     /.   .  ..
     / .  .  ..
    u .   .  . e
    .. .  . ./
     .. .  ./
```

b----c....f

We stop when no alternating (augmenting) paths can be found.

How do we find an alternating (augmenting) path?

  To find an augmenting path, use BFS from an unmatched
  vertex, exploring only non-matching/matching edges at
  alternate levels. In other words, when exploring the
  neighbours of a matched vertex, choose only matching edges
  (actually, there will always be exactly one matching edge to
  explore).

  This algorithm takes time M N (for each unmatched vertex, do
  a BFS to find an alternating path).

  Can we overlap the M BFS computations for the M unmatched
  vertex and do them in parallel?

  Initialize BFS queue with all M unmarked vertices, marking
  them as visited and at level 0. Now, we might have an
  augmenting path that passes from one unmarked vertex u via
  another unmarked vertex v. In this parallel BFS, we will not
  explore this path because when we reach v, we will find v
  already marked. However, notice that the suffix of the path
  starting from v is also an augmenting path on its own (the
  segment from u to v must be of even length, the total path is
  of odd length, so the suffix starting at v is a path of odd
  length that starts and ends at an unmarked vertex.) Thus, we
  can take the suffix starting at v as the augmenting path.

==
=======

Off with the head (INOI Training Camp, 2003)

The king has a large collection of ministers, all of whom he
believes to be corrupt. To teach them a lesson, he decides to
behead the most corrupt minister. To do this, he wants the prime
minister to conduct raids on all the ministers' houses and
identify the most wealthy one. The most wealthy minister will be
declared the most corrupt and beheaded.

There are 1,000,000 ministers and the king want the most corrupt
minister to be identified in 200 days. The prime minister cannot
raid raid all the ministers' houses within this time because he
has only three raiding teams at his disposal and each team takes
a full day to complete a raid. Thus he can raid only three
ministers' houses each day.

Instead, the prime minister decides to adopt a different
strategy. When the king meets the ministers, they all sit around
a big table. The prime minister decides that it is sufficient to
find one minister whose wealth is larger than that of his
neighbours. The king can then be convinced that this minister is
the most corrupt.

Devise a strategy for the prime minister to identify a minister
whose wealth is higher than that of his neighbours, raiding one
minister at one time.

Solution:

Probe three equally spaced points around the circle. Delete the
arc between the two smallest values and focus on the rest of the
circle which is a segment with three values that form a "tent".

```
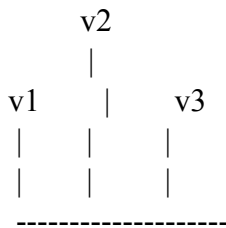            v2
            |
    v1      |      v3
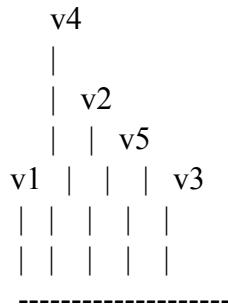    |    |     |
    |    |     |
     --------------------
```

Now probe the midpoints between (v1,v2) and (v2,v3), say v4 and
v5, respectively. Depending on the values here, you will find a
smaller tent.

Case 1: Shrink the tent to (v4,v2,v5)

```
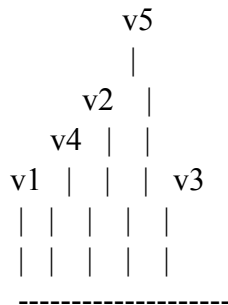          v2
       v4  |  v5
    v1 |  |  | v3
    | |  | |  |
    | |  | |  |
```

```
        --------------------
```

Case 2: Shrink the tent to (v1,v4,v2)

```
        v4
         |
         |  v2
         |  |  v5
    v1   |  |  |   v3
     |   |  |  |   |
     |   |  |  |   |
        --------------------
```

Case 3: Shrink the tent to (v2,v5,v3)

```
            v5
             |
        v2   |
      v4 |   |
    v1   |   |   |   v3
     |   |   |   |   |
     |   |   |   |   |
        --------------------
```

Repeat this till the tent consists of three consective positions.

Note: This is a library task, so a raid is conducted by calling a
      function.

```
======================================================================
==
```

Aliens (IOI 2007)

Mirko is a big fan of crop circles, geometrical formations of
flattened crops that are supposedly of alien origin. One summer
night he decided to make his own formation on his grandmother's
meadow. The great patriot that he is, Mirko decided to make a
crop formation that would have the shape of the shield part of
the Croatian coat of arms, which is a 5×5 chessboard with 13 red
squares and 12 white squares.

```
        xx  xx  xx
```

```
xx  xx  xx
 xx  xx
 xx  xx
xx  xx  xx
xx  xx  xx
```

Grandma's meadow is a square divided into N x N cells. The cell in the lower left corner of the meadow is represented by the coordinates (1, 1) and the cell in the upper right corner is represented by (N, N). Mirko decided to flatten only the grass belonging to red squares in the chessboard, leaving the rest of the grass intact. He picked an odd integer M >= 3 and flattened the grass so that each square of the chessboard comprises M x M cells in the meadow, and the chessboard completely fits inside the meadow.

After Mirko went to sleep, his peculiar creation drew the attention of real aliens! They are floating high above the meadow in their spaceship and examining Mirko's crop formation with a simple device. This device can only determine whether the grass in a particular cell is flattened or not.

The aliens have found one cell with flattened grass and now they want to find the center cell of Mirko's masterpiece, so that they may marvel at its beauty. They do not know the size M of each square in Mirko's formation.

Limits: $15 \leq N \leq 2 \times 10^9$. Can use the device (i.e. call the library) at most 300 times.

Solution:

Start from the given cell. You are inside a square. Query points that are $2^0, 2^1, 2^2, ..., 2^i$ to the right of this square till you find an unflattened cell.

```
xxxxxx     xxxxxx
xxxxxx     xxxxxx
xxxxxx     xxxxxx
xxxxxx     xxxxxx

 o------>o
   2^i
```

Note that you could not have crossed over a complete flattened square in this process.

You now do a binary search between the original point and the new point to find the right endpoint of this square.

Similarly, find the left endpoint and bottom endpoint. From left+right you know the size of the square, so you also know the top endpoint.

You need to find the centre of the overall pattern, which is of a fixed 5 x 5 size. Find the centre of the current square and walk up/left/right/down two squares at a time to do this.

===
==

Artemis (IOI 2004)

Zeus gave Artemis, the goddess of the wilderness, a rectangular area for growing a forest. With the left side of the area as a segment of the positive y-axis and the bottom side as a segment of the positive x-axis, and (0,0) the left bottom corner of the area, Zeus told Artemis to only plant trees only on integer coordinate points in the area. Artemis liked the forest to look natural, and therefore planted trees in such a way that a line connecting two trees was never parallel to x-axis or y-axis. At times, Zeus wants Artemis to cut trees for him. The trees are to be cut as follows:

1. Zeus wants at least a given number T of trees to be cut for him.

2. To get a rectangular football pitch for future football success, Artemis is to cut all trees within a rectangular area, and no trees outside of it.

3. The sides of this rectangular area are to be parallel to x-axis and y-axis.

4. Two opposite corners of the area trees must be located on

trees and therefore those corner trees are also cutstand in
two opposite corners of the area (and are included in it).

As Artemis likes the trees, she wants to fulfill these conditions
whilst cutting as few trees as possible. You are to write a
program that, given information of on the forest and the minimum
number T of trees to be cut, selects an area for cutting trees
for Artemis.

Limits:
  Grid is 64000 x 64000
  Number of trees is 10000 (was 20000 in IOI, but this seems wrong!)
  Memory is 16MB

Solution:

Easy algorithm stores the number of trees from (0,0) to (i,j) for
all trees (i,j), but this is not feasible given the memory limit.

Assume we are processing trees in ascending order of first
coordinates. Fix a tree (x,y) for which we want to calculate the
best rectangle with respect to all earlier trees to the left of
(x,y).

```
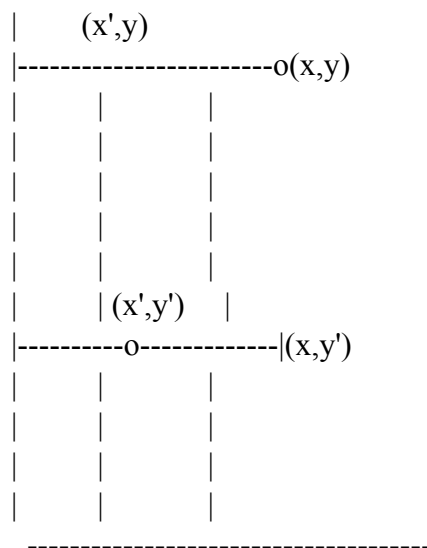|      (x',y)
|-----------------------o(x,y)
|      |        |
|      |        |
|      |        |
|      |        |
|      |        |
|      | (x',y')    |
|----------o-------------|(x,y')
|      |        |
|      |        |
|      |        |
|      |        |
   -------------------------------------
```

Keep two linear arrays
  - For each $1 <= j <= y$, $V[j] = 1$ if there is a tree at some (z,j)
  - For each $1 <= i <= x$, $H[i] = 1$ if there is a tree at some (i,z)

Prefix sums for H[] and V[] tell us number of trees to the left of some i and below some j.

For each tree (x",y") to the left of (x,y), assume we have stored the number of trees between (0,0) and (x",y"). Call this Between(0,0,x",y").

We can then get the number of trees between (x',y') and (x,y) as

  PrefixY(y) - [PrefixX(x') + PrefixY(y') - Between(0,0,x',y')]

We chedk if the number of trees in this rectangle is more than T and better than the best we know. If so, keep it, else throw it away.

In this way, after sorting the trees (N log N), we can process each tree in time N, so overall is N^2. The space we use is proportioanal to N (for each tree (x,y) store Between(0,0,x,y)).

==
=======
Paint (revisited)

A painter paints a strip N squares wide

```
        ----------------------
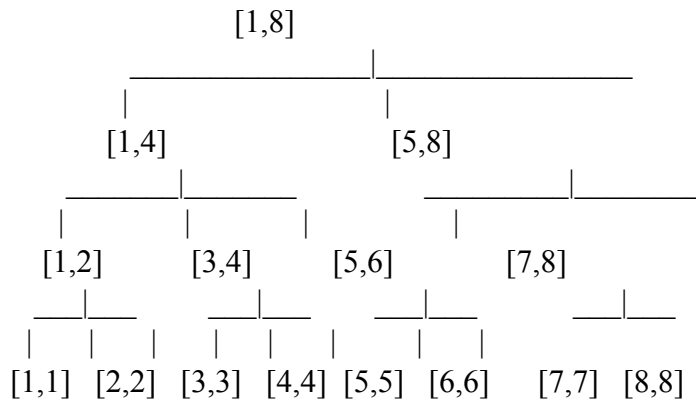        | 1 | 2 | 3 | .... | N |
        ----------------------
```

In one step, the painter paints a continuous strip from square i to square j, which we denote [i,j]. Periodically, we want to answer the following query:

  minlayers([i,j]) --- among all squares in the interval [i,j],
              report the minimum number of layers among
              the squares in this interval

How do we efficiently answer these queries?

Solution

Recall that we recorded the paint strokes in a segment tree.

```
                       [1,8]
          _____|_____
         |                               |
       [1,4]                           [5,8]
     _____|_____                  _____|_____
    |            |                 |            |
   [1,2]       [3,4]             [5,6]        [7,8]
   __|__       __|__             __|__        __|__
  |    |     |    |            |    |       |    |
[1,1] [2,2] [3,3] [4,4]     [5,5] [6,6]    [7,7] [8,8]
```

At each vertex v, when we apply a layer of paint, we record

   coats(v) = number of paint strokes that contain the entire
         interval Interval(v), but do not contain the entire
         interval Interval(parent(v)) spanned by the parent
         of v.

The number of coats at an interval is given by the value at
this node and the values of intervals that include this one
(these are above).

We need to store more information at each node to answer
minlayer queries. What is the structure of this information
and how do we use it to answer queries?

Suppose we keep the following additional information at each
node: for each interval v, how many full coats of v have come
from coats directly across v (but not the next bigger interval
spanning v) or by adding up coats over subintervals of v? For
children l and r of v, if we inductively know this value, add
the minimum of the two to coats(v).

   below(v) = number of paint strokes that span the
         interval Interval(v), either directly, or made up
         of smaller strokes.

      = coats(v) + min(below(leftchild(v)),below(rightchild(v)))

Now, with each stroke you walk down and update coats(v),

below(v) and then walk back up and recompute below(w) for each
w on the path above v.

==
=======

Yertle the Turtle (ACM)

Yertle, the turtle king, has 5607 turtles. He wants to stack up
these turtles to form a throne. Each turtle i has weight W[i]
and capacity C[i]. A turtle can carry on its back a stack of
turtles whose total weight is within C[i]. This should include
W[i], the weight of the current turtle, so assume that $C[i] >= W[i]$ always.

Find the height of the tallest turtle throne that Yertle can
build.

Solution:

Define residual capacity of turtle i as C[i] - W[i].

Claim: If there is a stack of height k, there is a stack of
  height k in which carrying capacities are sorted in descending
  order from bottom to top.

Define RC(i,k) to be the maximum of the following:

  Calculate each arrangement of height k using turtles 1..i
  Calculate the minimum residual capacity in this arrangement

Now

  RC(i+1,k) = max { RC(i,k),
            min{ (C[i+1]-W[i+1], (RC(i,k-1) - wt(i+1)) }
         }

The answer we want is the largest k for which RC[5607,k] is
positive.

This can be done in time N^2, which is feasible for N = 5607.

Memory? Naively, we need an 5607 x 5607 array. This is too
large. However, row RC[i+1,...] depends only on RC[i,...] so we
need to maintain only two rows of the matrix at a time.

===
=======
===
=======
         IOI Training Camp, Lecture Notes, Day 8

                  Sun 29 Jun 2008
===
=======
Session 1, 09:00--10:30
===
=======
Going to school

Father has two children who keep fighting. When they go to
school, they want to take routes on which they do not both have
to travel on the same road. Can the father send them to the same
school?

Example
        ->3
        / \
       /   \
     /--->2------------>6--
     /   \ \   \
    /   \ \   \
   H   ---->4   ---->S
    \   /   \   //
     \   /   \ //
      ---->1   -->5 /
          \   /

In general, are there k edge-disjoint paths from s to t? The

paths may meet at vertices.

Soln:

A uniform technique using network flows.

==
=======
Network flows

Given a network of pipes with a given capacity and two nodes,
source and sink, what is the maximum flow that can be passed from
source to sink?

```
          7
      -->u ------------->v
      / \         / \
     / 7  \      10 /   \5
    /      \     /     \
  source     \---- /      -->sink
    \      10   X      /
     \ 5        /\     /7
      \   <------/  \--> /
       -->w------------->z
            7
```
1. For each edge e, flow(e) <= capacity(e)

2. At each node, flow into the node is equal to the flow out of
   the node (conservation of flow).

In this case, the maximum flow is 12:
   send 5 along source->u->v->sink
   send 5 along source->w->z->sink.
   send 2 along source->u->v->w->z->sink

Observe that the edges (u,v) and (z,sink) together disconnect the
source from the sink. This is called a "cut set". Clearly, any
flow that goes from source to sink must go through this cut set
and cannot exceed the capacity of the cut, namely 14. There may
be many cut sets --- for instance, (source,u), (source,w) is also
a cut set, with capacity 12.

It is clear that the maximum flow in the graph cannot exceed the
capacity of the minimum size cut in the graph.

Claim: The maximum flow from source to sink is equal to the
  capacity of the minimum size cut in the graph.

How do we compute the maximum flow?

Suppose we start with a flow

```
      5   5   5         <--flow
  source---->o---->o---->sink
      5   10   5          <--capacity
```

Now, we select another path (ignoring directions) to augment the
flow.

An augmenting path is a path from source to sink where

-- forward edges, flow < capacity
-- backward edges, flow > 0

e.g. we may have a path that looks like

```
     4   6   3   2       <--flow
 source---->o----->o<-----o<------sink
     5   5   4   3        <--capacity
```

For each edge on this path, we compute leeway(e), defined as:

  if e is a forward edge : capacity(e) - flow(e)
  if e is a backward edge : flow(e)

```
     4   6   3   2       <--flow
source---->o----->o<-----o<------sink
     5   7   4   3        <--capacity
     1   1   3   1        <--leeway
```

Augment flow upto min leeway along the path. Here min leeway is
1, so you can augment flow. On reverse edges, cancel out the
existing flow, so reduce by the augmented amount.

After augmentation.

```
     5    7    2    3        <--new flow
 source---->o----->o<-----o<------sink
     5    7    4    3        <--capacity
```

Repeat the augmentation step till you cannot find an augmenting
path. (This algorithm is called the Ford-Fulkerson algorithm.)

Example:

```
         7
     -->u ------------->v
     / \          / \
    / 7  \      10 /   \5
   /     \      /    \
 source        \---- /        -->sink
   \      10    X       /
    \ 5         / \    /7
     \   <------/ \--> /
      -->w-------------->z
           7
```

Start with a flow

```
     5    5    5        <--flow
 source ---->u---->z---->sink
     7   10    7        <--capacity
```

Find an augmenting path, for example:

```
     5    0    0        <--flow
 source ---->u---->v---->sink
     7    7    5        <--capacity
     2    7    5        <--leeway
```

Add flow on this path

```
     7    2    2        <--augmented flow
 source ---->u---->v---->sink
     7    7    5        <--capacity
```

Now, another augmenting path (with backward edges):

```
         0   0   5   2   2      <--flow
  source ---->w---->z<----u---->v---->sink
         5   7   10  7   5      <--capacity
         5   7   5   5   3      <--leeway
```

Augmented flow

```
         3   3   2   5   5      <--flow
  source ---->w---->z<----u---->v---->sink
         5   7   10  7   5      <--capacity
```

Finally, we have an augmenting path

```
         3   3   5      <--flow
  source ---->w---->z---->sink
         5   7   7      <--capacity
         2   4   2      <--leeway
```

Augmented flow

```
         5   5   7      <--augmented flow
  source ---->w---->z---->sink
         5   7   7      <--capacity
```

This

Add s->w->z->u->v->t

How do we find augmenting paths?

  Use BFS. At each node:

  - take a forward edge if flow < capacity
  - take a backward edge if flow > 0

  If you reach the sink, you have found an augmenting path.

In the worst case, you have to do one iteration for each unit of
flow. Let F be the optimum flow. Then, we have F iterations,
each of which requires finding an s-t path (BFS/DFS), so it takes
mF time. (This assumes that flows are all integers).

Here is a worst case scenario

```
        10000    10000
          --->A---------->t
         /   \       -->
        /     \1   /
       s        -->/ 9000
         ----------->B
              9000
```

Here, if we always choose a path that includes the edge AB (or
BA), each step increases the max flow by exactly 1 unit, so it
takes 19000 iterations to terminate. (Of course, we could have
been lucky and chosen directly the upper and lower paths to
augment.)

Claim: If we always augment the flow by finding the shortest s-t
  path in terms of number of edges, the algorithm terminates in
  time O(mn). (We will not prove this). Notice that if we
  search for augmenting paths via BFS, we are guaranteed to find
  the shortest augmenting path at each stage.

Observe: If the capacities are integers, each time we augment the
  flow, we augment it by an integer. So, the max flow will
  always be integral.

===
=======

Returning to our problem "Going to school"

Solution:

Use max-flow. Put a capacity 1 on each edge. If max flow > 2,
there are at least 2 edge disjoint paths.

In general, the max flow gives the maximum number of edge
disjoint paths.

(The max flow-min cut theorem for 0/1 flows is called Menger's
  Theorem.)

Also, observe that since the flow increases with each iteration, we may be able to stop before the max flow is reached. For instance, if we are asked whether there are at least K edge-disjoint paths, we can stop when we attain a flow of K.

==
=======
Going to school, Version 2

Two children who fight as before. Roads are one way. Find a route from home to school that do not have any vertex in common.

Solution:

Split each vertex v into two copies, in(v) and out(v) and add an edge from in(v) to out(v) with a capacity of 1. Connect all incoming edges to v in original graph to in(v) and all outgoing edges from v in original graph to out(v). Any path that passes through v will create a flow of 1 and not permit any other path through v. So, again look for a max flow of at least 2.

==
=======
Evil intentions (IOI Camp Finals, 2005)

Tanmoy is searching for some special directed graphs so that he can force 90% of the submissions to time out. For each graph, his test data generator requires the vertices v1, v2, ... , vN to have indegrees i1, i2, ... , iN and outdegrees o1, o2, ... , oN.

For instance, suppose the sequence of indegrees required is 0,1,2,2 and the sequence of outdegrees required is 2,1,1,1. Here is a graph with the required properties.

```
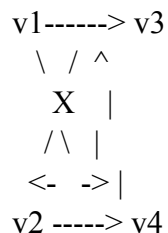        v1------> v3
         \ / ^
          X  |
         /\  |
        <-  ->|
        v2 -----> v4
```

Your task is to help him construct a graph that meets his requirements or report that no such graph exists, so Tanmoy has to find another way to achieve his evil goal.

Solution:

Take two copies of vertices {v1,v2,...,vn} and {v'1,v'2,...,v'n} and construct a bipartite graph where you connect vi to v'j for all j not equal to i. as a complete bipartite graph, add a source and sink and put capacities as follows:

```
   source --> vi   : capacity is outdegree(i)
   vi --> sink    : capacity is indegree(i)
   vi --> vj'     : capacity is 1


        --- v1--      v'1 ---
outdeg(1) /     \ \\ 1        \ indeg(1)
      / --- v2 \ \ -- v'2 --- \
      / /       \ \ 1       \ \
   source ----- v3   \ -- v'3 ------ sink
        \         1\          /
         \    ...    \  ...    /
outdeg(n) \           \     / indeg(n)
         --- vn       v'n --
```

Want a solution to maxflow = i1+i2+...in = o1+o2+...+on. (If these sums are not the same, there is no solution.)

If we find such a flow, the edges with flow 1 between vi and v'j define the graph you want.

Conversely, if there is a graph of the type you want, there will be a max flow as usual.

==
=======

Sails (IOI 2007)

A new pirate sailing ship is being built. The ship has N masts (poles) divided into unit sized segments - the height of a mast is equal to the number of its segments. Each mast is fitted with

a number of sails and each sail exactly fits into one
segment. Sails on one mast can be arbitrarily distributed among
different segments, but each segment can be fitted with at most
one sail.

Different configurations of sails generate different amounts of
thrust when exposed to the wind. Sails in front of other sails at
the same height get less wind and contribute less thrust. For
each sail we define its inefficiency as the total number of sails
that are behind this sail and at the same height. Note that "in
front of" and "behind" relate to the orientation of the ship: in
the figure below, "in front of" means to the left, and "behind"
means to the right.

The total inefficiency of a configuration is the sum of the
inefficiencies of all individual sails.

Example

```
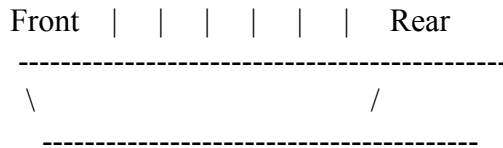        /|
       / 0|
       ---+
        /|  /|        /|
       / 2| / 1|      / 0|
       ---+ ---+      ---+
     /|   |   |      /|  /|
    / 2|  |   |     / 1| / 0|
    ---+  +   +     ---+ ---+
      |  /|   |      |  /|
      | / 1| |       | / 0|
      +  ---+ +       +  ---+
     /|   |   | /|  /|   |
    / 2|  |   | /1|  /0|   |
    ---+  +   + --+  --+   |
```

```
   Front   |   |   |   |   |   |    Rear
    -----------------------------------------------
      \                              /
       -----------------------------------------
```

This ship has 6 masts, of heights 3, 5, 4, 2, 4 and 3 from
front (left side of image) to back. This distribution of sails
gives a total inefficiency of 10. The individual inefficiency
of each sail is written inside the sail.

Write a program that, given the height and the number of sails on
each of the N masts, determines the smallest possible total
inefficiency.

Solution:

Observation 1:

The inefficiency of a level is determined by the number of sails
at that level. The order of the masts does not matter, so we can
rearrange them in ascending order of height.

Greedy strategy:

Place the sails from front (shortest) mast to rear (tallest)
mast. For each new mast, place each sail on the level from the
bottom that has least number so far. (As you go back and the
masts increase in height, so new levels have count 0.)

Is this greedy strategy correct?

Given any optimal placement, we can show that we can transform
step by step it into something that the greedy strategy will
produce. (This is the standard way to prove a greedy algorithm
is correct --- "exchange" argument.)

Initially, all levels have 0. So, any placement of the first
mast is consistent with the greedy suggestion.

Let us assume that upto the first K masts, we have perturbed the
optimal solution to look like greedy. We now show that we can
rearrange the (K+1)st mast to also be consistent with greedy
without losing optimality.

Why is the (K+1)st mast not compatible with greedy?

There is (at least) one mast that is in the wrong place according
to greedy, and the greedy slot is vacant.

       l1   O   r1

      l2   *   r2

  <--------------------> <------------------>
   1 to K is greedy   K+1 K+2 onwards is not greedy

Assume that $l1 < l2$, so we want to move the mast from * to o.

Suppose we move the mast from l2 to l1.

The new inefficiency is

  old + (l1 - l2) + (r1 - r2)

This is less than or equal to old provided

  (r1 - r2) <= (l2 - l1)

If this inequality holds, we are done.

What if this inequality does not hold?

  r1 - r2 > l2 - l1 > 0

Therefore, r1 is at least 1. Since $r1 - r2 > 0$, there is at
least one position where r1 has a sail and r2 does not. We move
one sail from r1 to r2.

Now, the new cost is

  old + (l1 - l2) + (r1 - r2) + ((l2 + r2) - (l1 + r1))

which is just the same as old.

Implementing greedy:

Maintain a min-heap with pairs of the form (level, sails). When
we reach a new mast, insert (l,0) for each newly added level.

This will take time O(number of sails * log (levels)), which gets
40% of the marks.

A more efficient implementation:

Store the levels sorted in descending order. Instead of keeping
the absolute value of sails per level, keep the difference of
each level with respect to the previous level in the sorted
sequence.

e.g., if the levels are ordered as

    10 10 9 8 8 7 7 7 7 0 0

we store the differences

    10 0 -1 -1 0 -1 0 0 0 -7 0

When we process a new mast, we first append the new empty levels
at the right of this list.

Now we want to add K sails to the levels with the lowest
occupancy, namely the last K. This means we have to update the
last K difference values.

In general, to update a segment [i,j], we only need to change the
values at the left and right end --- decrease the difference i -
(i-1) by 1, or increase the entry at i by 1, and increase the
difference at (j+1) - j by 1, or reduce the entry at j+1 by 1.
If one endpoint of the segment to be updated is the right end, we
only have to update the left endpoint.

For instance, if we add 6 sails from the right to the sequence
above, we get

```
10  0 -1 -1  0  0  0  0  0 -7  0
             <-------------------->
                 updated levels
```

This difference list corresponds to the updated level values

```
10  10  9  8  8  8  8  8  8  1  1
```

after adding 1 to the last 6 entries.

What if we added only 4 entries, not 6? The new difference list would have been

```
10  0 -1 -1  0 -1  0  1  0 -7  0
                  <----------->
```

This corresponds to the updated level values

```
10  10  9  8  8  7  7  8  8  1  1
```

which correctly captures the fact that the last 4 entries have been incremented, but this list is not in sorted order.

The problem is that the updated segment ended in the middle of a "level group" of 7's. It is not difficult to see that the only values that are not in sorted order are the elements at the right end of the level group that was only partially incremented.

When we update part of level group, we have to shift the elements from the right of the group to the left end. In this case, the final sorted sequence we should have is

```
10  10  9  8  8  8  8  7  7  1  1
```

whose difference sequence is

```
10  0 -1 -1  0  0  0 -1  0 -6  0
```

This corresponds to taking the level group of 7's in the original sequence and incrementing a segment [6,7] by 1, so we increase the value at position 6 and decrease the value at position 7+1 = 8, as described for a general [i,j] segment earlier.

```
                   level group
                  <------------>
    10   0  -1  -1   0  -1   0   0   0  -7   0
                  <---->        <---->
                  update        update


    10   0  -1  -1   0   0   0  -1   0  -6   0
```

This gives us the overall update procedure for adding K sails on
a mast:

  Count the last K levels from the right. If this ends at the
  boundary of a level group, do a simple update for the interval
  [N-K+1,N].

  Otherwise, update everything to the right of the incomplete
  level group as usual and separately update the left end of the
  level group.

How do we find the end point of a level group? A level group
[A,B] consists of a contiguous interval of 0's, so we need to
find the next non-zero value to the left and right of a given
position K from the right. To get full credit, this has to be
done using a segment tree. For each interval of the sequence of
levels, store the position of first nonzero position from the
right end of an interval and the first nonzero position from the
left end of an internval. Using this, we can find the endpoints
of the level group containing a given position K in time
proportional to log(number of levels). This segment tree can
also be updted in time proportional to log(number of levels).

Overall, the complexity is then O(number of masts * log (max levels)).

==
=======
Flood (IOI 2007)

In 1964 a catastrophic flood struck the city of Zagreb. Many
buildings were completely destroyed when the water struck their
walls. In this task, you are given a simplified model of the city
before the flood and you should determine which of the walls are

left intact after the flood.

The model consists of N points in the coordinate plane and W walls. Each wall connects a pair of points and does not go through any other points. The model has the following additional properties:

- No two walls intersect or overlap, but they may touch at endpoints;

- Each wall is parallel to either the horizontal or the vertical coordinate axis.

Initially, the entire coordinate plane is dry. At time zero, water instantly floods the exterior (the space not bounded by walls). After exactly one hour, every wall with water on one side and air on the other breaks under the pressure of water. Water then floods the new area not bounded by any standing walls. Now, there may be new walls having water on one side and air on the other. After another hour, these walls also break down and water floods further. This procedure repeats until water has flooded the entire area.

An example of the process is shown in the following figure.

```
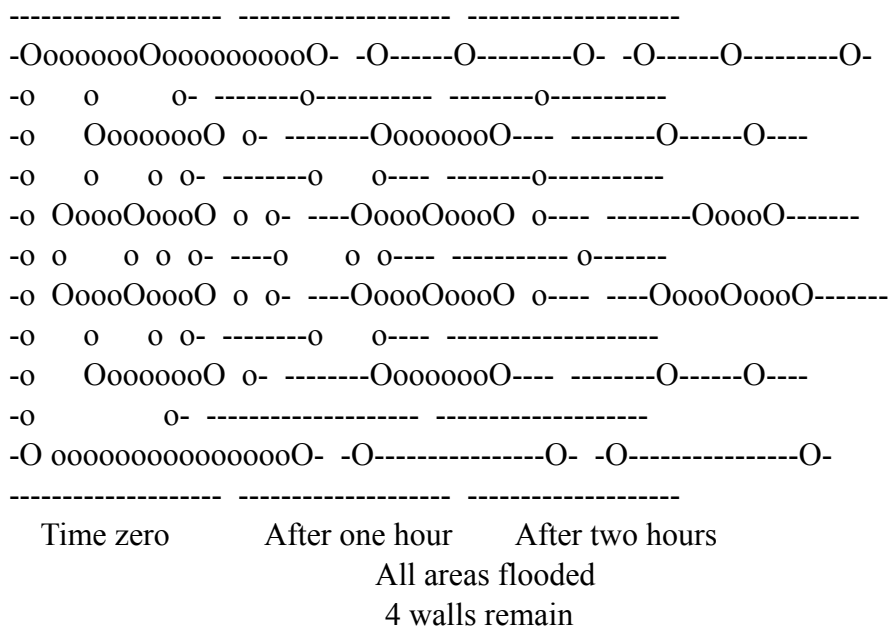-------------------- -------------------- --------------------
-OoooooooOooooooooooO- -O------O---------O- -O------O---------O-
-o      o     o- --------o----------- --------o-----------
-o    OoooooooO o- --------OoooooooO---- --------O------O----
-o    o    o o- --------o    o---- --------o-----------
-o OoooOoooO o o- ----OoooOoooO o---- --------OoooO-------
-o o    o o o- ----o    o o---- ----------- o-------
-o OoooOoooO o o- ----OoooOoooO o---- ----OoooOoooO-------
-o    o    o o- --------o    o---- --------------------
-o    OoooooooO o- --------OoooooooO---- --------O------O----
-o         o- -------------------- --------------------
-O oooooooooooooooooO- -O---------------O- -O---------------O-
-------------------- -------------------- --------------------
    Time zero        After one hour     After two hours
                        All areas flooded
                        4 walls remain
```

Solution:

Construct a graph in which nodes are regions (maximal set of connected cells). Connect two regions by an edge if they share a wall. Add the outer region as a node and connect it to all regions initially exposed to water.

The solution is then obtained by running BFS on this graph, but the difficulty is to construct the graph to begin with!

Another approach is to, walk systematically around the outermost boundary, say clockwise

  Pick an extreme edge
  * if there is a left edge from a vertex, take it
  * else if there is an up from a vertex, take it
  * else if there is a down edge from a vertex, take it
  * else go back

If an "external" wall has water on both sides, we will visit it twice in this traversal. Any wall we see only once will collapse. So, we maintain a count of how many times we have seen each wall as we go. Those that are visited only once will collapse in this round and can be eliminated.

We repeated traverse the outer wall removing walls till no more walls collapse.

==
=======

Pairs (IOI 2007)

Mirko and Slavko are playing with toy animals. First, they choose one of three boards given in the figure below. Each board consists of cells arranged into a one, two or three dimensional grid.

  1 dimensional: Line of length M
  2 dimensional: Square grid of size M x M
  3 dimensional: Cubic grid of size M x M x M

Mirko then places N little toy animals into the cells.

The distance between two cells is the smallest number of moves that an animal would need in order to reach one cell from the other. In one move, the animal may step into one of the adjacent cells (connected by line segments in the figure).

Two animals can hear each other if the distance between their cells is at most D. Slavko's task is to calculate how many pairs of animals there are such that one animal can hear the other.

Given the board type, the locations of all animals, and the number D, find the desired number of pairs.

Limits:

$N \leq 10^5$
$D \leq 10^8$
For $B = 1$, $M \leq 75 \times 10^6$
For $B = 2$, $M \leq 75 \times 10^3$
For $B = 3$, $M \leq 75$

Solution:

1 dimensional:

Sort the animals by increasing order of position (need only positions where animal occur, not all possible positions).

Use a sliding window to count for each point, how many animals are within distance D from the left. Each pair is counted once in this process.

2 dimensional:

Consider a pair of points (Px,Py) and (Qx,Qy). Want to identify if they are distance D apart walking along the grid.

Distance is $|Px - Qx| + |Py - Qy|$

Possibilities:

1. $(Px - Qx) + (Py - Qy) = (Px + Py) - (Qx + Qy)$
2. $(Qx - Px) + (Py - Qy) = -(Px - Py) + (Qx - Qy)$

3. (Px - Qx) + (Qy - Py) = (Px - Py) - (Qx - Qy)
4. (Qx - Px) + (Qy - Py) = -(Px + Py) + (Qx + Qy)

Notice that $|Px - Qx| + |Py - Qy|$ is always the max of these four quantities.

Let us define

  Pd1 := Px + Py   Qd1 := Qx + Qy
  Pd2 := Px - Py   Qd2 := Qx - Qy

Now, Distance = max{Pd1 - Qd1, Pd2 - Qd2, Qd2 - Pd2, Qd1 - Pd1}

which is the same as

      max {|Pd1 - Qd1|, |Pd2 - Qd2|}

With this transformation of coordinates, we have essentially mapped a 2 dimensional problem to two 1 dimensional problems.

Henceforth, we represent each input point (Px,Py) as (Pd1, Pd2). We sort these points along the (new) x-coordinate and use a sliding window to fix an interval of points at distance D along the x-axis for each x value.

Let P be the point at the head of the sliding window. For all points in this interval, we also need to consider points Q whose y coordinates satisfy $|Pd2 - Qd2| <= D$.

```
|         x
|     . .x............
|    x.     x x  . ^
|     .  x       . |D
|   x .   x    x . v
|    x .          P ^
| x    .           . |D
|      .          . v
   -------------------------
       <-------D------>
```

For this, we keep all points between head and tail in a
binary indexed tree (or segment tree) based on y-coordinate.
We can then query this tree to find out how many points lie
between Pd2-D and Pd2+D

As we move the sliding window along x to the right, we have
to delete the corresponding y-coordinates from the segment
tree and add the y-coordinates for the new points we add.

Note that M <= 75000, so we can keep a segment tree in which
we keep track of all intervals, not just coordinates that
correspond to actual points.

3-D:

Similar to the 2-D case, but transform into four coordinates.

In principle, the 3D solution can be specialized to the 2D and 1D
case by setting all z-coordinates (or all y and z coordinates) to
a constant value, but the varying dimensions in the three case

==

Roads (APIO 2008)

The Kingdom of New Asia contains N villages connected by M
roads. Some roads are made of cobblestones, and others are made
of concrete. Keeping roads free-of-charge needs a lot of money,
and it seems impossible for the Kingdom to maintain every road. A
new road maintaining plan is needed.

The King has decided that the Kingdom will keep as few free roads
as possible, but every two distinct villages must be connected by
one and only one path of free roads. Also, although concrete
roads are more suitable for modern traffic, the King thinks
walking on cobblestones is interesting. As a result, he decides
that exactly K cobblestone roads will be kept free.

For example, suppose the villages and the roads in New Asia are
as in Figure 1a. If the King wants to keep two cobblestone roads
free, then the Kingdom can keep roads (1,2), (2,3), (3,4), and

(3,5) free as in Figure 1b. This plan satisfies the King's criteria because (1) every two villages are connected via one and only one path of free roads, (2) there are as few free roads as possible, and (3) there are exactly two cobblestone roads: (2,3) and (3,4).

```
 1-------- 2            1-------- 2
  .    . \               .
   .   .  \               .
    . .    \               .
     3 . . . . 4            3 . . . . 4
      \   /                  \
       \ /                    \
        5                      5
```

Solution:

We have a graph with edges of two types, say coloured black (concrete) and blue (cobblestones). We want a spanning tree with exactly K blue edges.

We first delete the blue edges and compute the connected components. In any spanning tree, these components have to be connected by blue edges. If there are more than K+1 components, there can be no spanning tree using exactly K blue edges.

What if we have <= K components?

Recall Kruskal's algorithm for a minimum cost spanning tree. We set all weights to 1 and run Kruskal's algorithm as follows:

1. First add only blue edges that connect disjoint components (those that joined by only black edges).

2. Once we have connected all components, we continue to add blue edges, till we have added K edges.

3. Once we have put in exactly K blue edges, we stop using blue edges and start adding only black edges.

This is correct because Kruskal's algorithm will find the

spanning tree for any permutation of edges that are in sorted
order. In particular, if all edges have equal weight, you can
provide the edges to in any order. If at any stage we find that
we cannot continue, this means that there is no possible spanning
tree with exactly K blue edges and we stop and report failure.

===
=======

DNA (APIO 2008)

One interesting use of computer is to analyze biological data
such as DNA sequences. Biologically, a strand of DNA is a chain
of nucleotides Adenine, Cytosine, Guanine, and Thymine. The four
nucleotides are represented by characters A, C, G, and T,
respectively. Thus, a strand of DNA can be represented by a
string of these four characters. We call such a string a DNA
sequence.

It is possible that the biologists cannot determine some
nucleotides in a DNA strand. In such a case, the character N is
used to represent an unknown nucleotides in the DNA sequence of
the strand. In other words, N is a wildcard character for any one
character among A, C, G or T. We call a DNA sequence with one or
more character N an incomplete sequence ; otherwise, it is called
a complete sequence. A complete sequence is said to agree with an
incomplete sequence if it is a result of substituting each N in
the incomplete sequence with one of the four nucleotides. For
example, ACCCT agrees with ACNNT, but AGGAT does not.

Researchers often order the four nucleotides the way we order the
English alphabets: A comes before C, C comes before G, G comes
before T. A DNA sequence is classified as form-1 if every
nucleotide in it is the same as or comes before the nucleotides
immediately to its right. For example, AACCGT is form-1, but
AACGTC is not.

In general, a sequence is form-j , for $j > 1$, if it is a
form-$(j-1)$ or it is a concatenation of a form-$(j-1)$ sequence and
a form-1 sequence. For example, AACCC, ACACC, and ACACA are
form-3, but GCACAC and ACACACA are not.

Again, researchers order DNA sequences lexicographically the way
we order words in a dictionary. As such, the first form-3

sequence of length 5 is AAAAA, and the last is TTTTT. As another example, consider the incomplete sequence ACANNCNNG. The first seven form-3 sequences that agree with it are:

```
ACAAACAAG
ACAAACACG
ACAAACAGG
ACAAACCAG
ACAAACCCG
ACAAACCGG
ACAAACCTG
```

Given an incomplete sequence of length M, and two values K and R, the task to find the Rth form-K sequence that agrees with the given incomplete sequence.

Here M <= 50000, K <= 10, R <= 2^{12}.

Solution:

A form-K sequence can be decomposed into at most K blocks, where block has letters in non-decreasing order.

To start with let us write a recurrence to compute the number of form-K sequences of a given length, without any constraints.

For $1 <= i <= m$, $1 <= j <= K$, X in {A,C,G,T}, let

   f(i,j,X) denote the number of form-K sequences from positions
        i to M starting with X.

There are two cases to consider:

- The letter at position i+1 is strictly smaller than X. If so,
   a new block starts at i+1 so we need to consider form-(j-1)
   sequences from i+1

- The letter at position i+1 is greater than or equal to X. If so,
   the block at i continues into i+1, so we need to consider form-j
   sequences from i+1.

This gives us the recurrence

$$f(i,j,X) \;=\; \text{sum} \;\; f(i+1,j-1,Y) + \;\; \text{sum} \;\;\; f(i+1,j,Y)$$
$$\qquad\qquad\quad Y < X \qquad\qquad\quad Y >= X$$

Now, how do we count the number of form-K sequences compatible
with a given pattern? The pattern fixes values for certain
positions. Suppose the p[i], the ith position in the pattern, is 'A'.
Then, for all j

  f(i,j,X) = 0 for X not equal to 'A'

In this way, we fill in 0's for all values f(i,j,X) such that
p[i] is not N and p[i] is not equal to X. We then process the
recurrence as before to get the total number of form-K sequences
compatible with the pattern.

Now, how do we find the Rth such sequence?

The total number of form-K sequences is, in general, the sum

  f(1,j,'A') + f(1,j,'C') + f(1,j,'G') + f(1,j,'T')

Let these values be n_A, n_C, n_G and n_T, respectively.
Clearly, the first n_A sequences in lexicographic order start
with A, the next n_C sequence start with C, ... If R <= n_A, we
know the first letter is A. If n_A < R <= n_A + n_C, the first
letter is C, and so on.

Suppose that n_A < R <= n_A+n_C. This fixes the first letter to
be C and we now look for the sequence with index R_1 = (R - n_A)
starting with C.

As before, we know that n_C is the sum of four terms, n_CA, n_CC,
n_CG and n_CT. Again, by checking if R1 <= n_CA or n_CA < R1 <=
n_CA+n_CC, ... we can determine the second letter of the
sequence.

This yields a new value R2 with which we can compute the third
letter, and so on.

===
=======
Zoo (APIO 2007)

(APIO 2007)

A zoo has N (<= 10,000) different animals in cages, arranged in a
circle. C (<= 50,000) schoolchildren visit the zoo. Each child
stands in a position where he can see 5 consecutive cages. Each
child has a set of animals that he fears and a set of animals
that he likes. A child is happy if

(a) there is some animal in his view of 5 cages that he likes

OR

(b) some animal that he fears has been removed from his view of 5
    cages

You are given the locations of the C children and their fears and
likes. What is the maximum number of children who can be made
happy by removing some of the animals from their cages?

Example:

Child	View	Fears	Likes
Alex	2,3,4,5,6	Cage 4	Cages 2,6
Polly	3,4,5,6,7	Cage 6	Cage 4
Chaitanya	6,7,8,9,10	Cage 9	Cages 6,8
Hwan	8,9,10,11,12	Cage 9	Cage 12
Ka-Shu	12,13,14,1,2	Cages 12,13,2	-

Here, removing the animals in cages 4 and 12 would make Alex and
Ka-Shu happy. Chaitanya is already happy because he can see 6
and 8 which he likes. However, Polly and Hwan have become unhappy.

In this example, you can remove the animal in cage 13 and make
all 5 happy.

Solution

For each cage, either we remove the animal or not. Thus, there

are 2^N possible combinations to try out. We optimize the search
using dynamic programming by focusing on intervals of length 5,
since this is the view of a single child.

Initially, let us assume the cages are in a line 1,2,...,N,
rather than in a circle.

Define:

  View(i) : Number of children who can see cages 1,2,...,i

  End(i) : Children whose view ends with cage i

  Happy(i,Ci,Ci-1,Ci-2,Ci-3,Ci-4) : Maximum number of children
      from View(i) who are happy, given that the last five cages
      in 1..i are set as (Ci,Ci-1,Ci-2,Ci-3,Ci-4) --- each Cj is
      either set to empty or full.

We want to extend Happy(i,...) to

  Happy(i+1,Ci+1,Ci,Ci-1,Ci-2,Ci-3).

The "new" children we consider at this stage are those in
End(i+1), whose view ends at i+1. For each choice of
(Ci+1,Ci,Ci-1,Ci-2,Ci-3), we can compute the number of "new"
children who are happy with such a choice.

Notice that choosing (Ci+1,Ci,Ci-1,Ci-2,Ci-3) fixes four cages
with respect to i. The cage that is not fixed is Ci-4, which
could be empty or full. Thus, if we fix the last 5 cages upto
i+1 as (Ci+1,Ci,Ci-1,Ci-2,Ci-3), we can lookup

  max{ Happy(i,Ci,Ci-1,Ci-2,Ci-3,empty),
      Happy(i,Ci,Ci-1,Ci-2,Ci-3,full) }

to know how many "old" children are kept happy by this choice.

We have to add to this the number of "new" children who have been
made happy by the choice for Ci+1.

We choose a value of (Ci+1,Ci,Ci-1,Ci-2,Ci-3) that maximizes the
sum of "new" and "old" children made happy by this choice.

Thus, at level i+1, we have to compute $2^5 = 32$ combinations of

  Happy(i+1,Ci+1,Ci,Ci-1,Ci-2,Ci-3)

Each child j's happiness is checked for the five values of i
corresponding to his view. Thus, overall, we will check each
child's happiness $5*2^5$ times. This gives an overall complexity
of $(N + C)*2^5$.

What happens when we move from a line to a circle? For positions
N-3,N-2,N-1,N, we have to take into account the choices made for
positions 1,2,3,4 when calculating "old" values. Thus, for
example, when we do N-1, we have to count a child as "old" if the
child is happy with the choices made for cages N-5,N-4,N-3,N-2 or
for the choices made right in the beginning for cages 1,2, since
the child's view could extend from N-1 in either direction.

In other words, we do a copy of the line calculation above for
each possible configuration of (C1,C2,C3,C4) and take this stored
configuration into account when computing Happy(i,....) for i =
N-3,N-2,N-1,N. This adds a factor 2^4 to the estimate above.

==
==

==
=======
        IOI Training Camp, Lecture Notes, Day 9

                Mon 30 Jun 2008
==
=======
Session 1, 09:00--10:30
==
=======
Triangulation

We consider a special type of polygon in which the vertices form
an upper and lower chain, connected at the ends, so that both
chains move monotonicaly from left to right.

The aim is to triangulate the polygon --- that is, add diagonal

edges between vertices that lie entirel within the polygon so
that the interior is completely divided into triangles.

Vertices form two chains

```
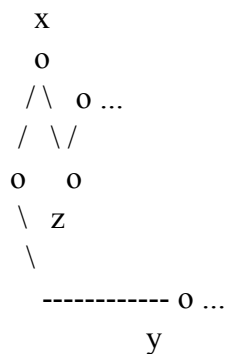    ---->
       o
     o--/ \-o--o
    /         \
   o    o    o
    \  /\  /
     --o   \/
           o
    ---->
```

The only constraint is that the horizontal order of top and
bottom chains go left to right. The top chain could come down
below the highest point on the bottom chain.

```
      o
     / \        ...
    / o -   /
   / /\ \ / / ...
  / -- \  o /
 o /    \  /
        \ /
        \/
        o
```

One standard difficulty with triangulating the polygon is the
fact that an upper vertex may "hang down" and block a diagonal
from an upper vertex to its left to a vertex below and to its
right.

```
    x
    o
   /\  o ...
  / \/
 o   o
  \ z
   \
     ----------- o ...
          y
```

Cannot draw a diagonal from x to y because it goes out of the
polygon to the right of z.

Solution:

If we can find a good diagonal, we can try divide and conquer.

Problems:

a. The diagonal we pick may not split the polygon well
b. How do we go about efficiently finding a diagonal that belongs
   to the triangulation.

Instead, scan from left in sorted order across both top and
bottom chains. Consider the first four points from the left.

Problem case:

     o

        ... lots more top points ...
     o

    o
   o

            o
             leftmost
             bottom point

Call this ascending slope on the left a "mountain".

As we reach each point p_i in sorted order (across both chains),
we need to consider cases

  e.g. p_i on the top chain, p_i on the bottom chain,
    p_i cause curve to go up, p_i causes curve to go down

Also need to identify an invariant that holds before and after
processing each point p_i.

Some observations.

  If p_i is on the up chain and the "mountain" continues to grow,
  do nothing and continue

  If p_i is on the up chain and the "mountain" tapers off, try to
  connect p_i to points on the mountain slope to the left.

  If p_i is from the lower chain, we try to connect it to the
  "mountain". There cannot be a "hanging" point from above that
  obstructs this because we are considering points in sorted
  order.

  Invariant:

  The unprocessed points consist of a mountain on the top chain.
  The edges on the mountain may either be original polygon edges
  or triangulation edges that we have added.

  Some points lie to the left of the mountain, but all these
  points are already triangulated.

            these   o
            points
            have been
            processed   o
            <-----
                    o
                o
                s

        s is the bottom left corner of the mountain

Algorithm:

  sort points from left to right

  for each point p_i do{

    if p_i is from the top chain {

if the slope is increasing, push it onto the stack of
"mountain points"

if the slope decreases{

repeat {
pop highest unprocessed point p_j off the "mountain"

connect p_i to p_j if possible
(this is possible if the slope from p_j to p_i is
less than the slope from p_j to the previous point
on the mountain)
} until top of stack cannot connect to p_i

update the mountain to include the last p_j to p_i
edge added
}
}
else if p_i from the bottom chain {

connect p_i to all points on the mountain

(this is always possible because p_i is the next
point in sorted order, so there cannot be any top
chain points "hanging" down blocking any of these
diagonals.)

set p_i to be the start point s of a new mountain.
}
}

Time taken:

Sorting takes O(n log n)

After sorting, each point enters the stack once and leaves
once, so the processing time is O(n).

Generalize

What if the polygon is not the special case above.

Can we add edges to break up the polygon to reduce it several new
polygons, each of which has the structure above?

```
         o
        / \
       /   \
      o ...... o
       \   /
        o...o
        /   \
       /     \
      o ........o
       \     /
        o...o
        /   \
       /     \
      o......... o
       \     /
        o---o
```

We sweep a vertical line from left to right across the polygon.
Because the polygon can "bend back", the portion to the left of
the line is, in general, a set of disjoint chunks, as shown
below.

```
          |/
          /
 chunk 1     /|
        o |
          \|
           \
           |\
           | o
           |/
           /
          /|
 chunk 2    / |
        o |
          \ |
          \|
```

```
        \
```

Case analysis:

Let us classify the points that we meet as we sweep our line from left to right across the polygon.

```
   /
1. o  P      (P is the polygon, to the right of the new vertex)
   \
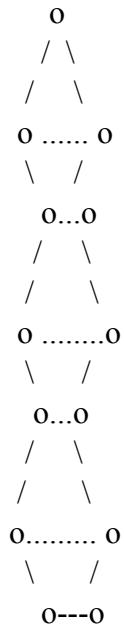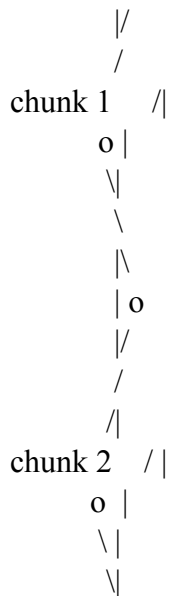

    /
2. P o    polygon to the left
    \        (this case does not arise in two-sorted-chain case)


   \
3. P o    polygon to the left
   /         (in the earlier case, this is the unique end point)


   \
4.   o P   polygon to the right
   /         (this case does not arise in two-sorted-chain case)



5.   o---
    / P
```

Cases 2 and 4 are problem cases.

How do we identify which case holds?

  a. Check slope with previous and next point.

  b. Also need to know which side the polygon is.

    The polygon is initially provided as a cyclic sequence of edges around the boundary.

    If we know the orientation of the polygon with respect to any one edge, as we traverse the boundary, we can fix the

orientiation for every edge.

How do we find the orientation of one edge? Take the
leftmost point overall. It must be of type 1, so we can
orient its outgoing edges.

Invariant:

The points to the left of the current point form a collection of
disjoint blocks.

```
  /
 ./
  .
  \
   \
```

Type 1 happens between chunks. Opens a new chunk

Type 2 happens inside a chunk. Connect it to points on its
    left. How?

Type 3 closes a chunk

Type 4 merges chunks.

What does the rightmost point in each chunk correspond to?

 Can be of type 4 or type 5

Updated invariant:

 Maintain chunks with rightmost point

  -- top and bottom incomplete edges that intersect vertical
     line where we are currently

  -- assume there is exactly one "unsatisfied" rightmost point,
     either type 4 or 5

When we add a type 2 point, connect it to the rightmost
unsatisfied point.

-- This is always possible (rightmost point is of type 4 or
   type 5).

-- What if there other type 4 points in the chunk? Assume
   inductively that there is only one "unsatisfied" type 4
   point in the chunk

When we add a type 3 point, we close a chunk.

When we add a type 4 and merge chunks, we get a bigger chunk?

-- The new type 4 point will be the rightmost "unsatisfied"
   point. We connect the new point to the rightmost
   unsatisfied chunk in the two chunks that just got merged to
   eliminate them from consideration. Since there is only one
   bad point in each,

When we add a type 5 point?

-- If there is rightmost "unsatisfied" type 4 point in this
   chunk, connect the new point to it. The new point is now
   the rightmost point (of type 5).

===
=======
Session 2, 11:00--12:30
===
=======
Placing guards

A polygonal museum. Place guards at some corners so that between
them they can see the whole polygon.

Solution:

Difficult to get the optimum. Suppose you want to place no more
than n/3 guards for n corners.

Triangulate the polygon. Colour the vertices of each triangle
with three colours, red, blue and green. This can be done
systematically so that each triangle has three colours and

overall, no two adjacent vertices have the same colour.

Now, in this global colouring, at least one colour appears on <= n/3 vertices. If we place guards at vertices of this colour, they can see all triangles and hence the entire polygon

How do we guarantee that we can 3-colour the polygon by 3-colouring each triangle.

In any triangulation, there must be one triangle in which two edges are two consecutive edges on the polygon and the third edge is a diagonal. Call it an "ear" triangle.

```
      o-------
     / .
    /  .
   o  ear.   rest of the polygon
    \  .
     \ .
      o------
```

Remove the ear and have a residual polygon with n-1 vertices using the diagonal of the ear as an edge. Recursively 3-colour this smaller polygon. This would assign 2 colours to the diagonal of the original ear, leaving one free colour for the vertex that we left out.

Does every polygon have an ear?

For a polygon with n points the triangulation has n-3 diagonals and n polygon edges dividing the polygon into n-2 triangle. Each polygon edge takes part in exactly one triangle. Also, unless we have a triangle, no triangle has 3 polygon edges. Since we have n-2 triangles and n polygon edges, there must be at least two triangles with two polygon edges, so every polygon with 4 or more sides has at least two ears.

How long does it take to find a ear?

Assume we have a triangulation in n log n time.

If we walk around the boundary, it take linear time. This would take time O(n) for each recursive step, so this would be n^2

overall, since the size decreases as n, n-1, ...

Is there a faster way to do this?

Suppose we construct a graph corresponding to the triangles:

  Vertices are triangles (n-2 vertices)
  (t1,t2) is an edge if t1 and t2 share an edge

Each edge cuts one diagonal. There are n-3 diagonals, so this graph is connected with n-2 vertices and n-3 edges. This means it is a tree.

The ears are degree 1 nodes in this tree. There must be at least two degree one nodes, hence at least two ears.

Having found a leaf, we delete it and look for another a leaf.

Having observed that the triangulation forms a tree, we can now argue that we can recursively construct a 3-colouring by starting witn any triangle in the triangulation.

===
=======
Median lines

We are given N (an odd number) of straight lines, $\{y = m_j x + c_j\}$, each with a distinct slope.

If we start at the y-axis and draw a vertical line, it will cut all N lines. We identify the median line that it crosses. As the vertical line sweeps right, the median line will change. Construct the sequence of median lines that we see as the vertical line sweeps to the right.

Solution:

Sort all N^2 points of intersection from left to right.

Start with the median line. Find the left most intersection on this line to the right of the y-axis. At this point the median switches. On the new line, again find the leftmost intersection

to its right.

Can we find a solution better than N^2?

Hint: Eliminate intersections that cannot contribute to the
    median.

Think about the problem.

Ramesh Hariharann's solution:

(I realized too late that this is a little complicated. Here is
a very rough outline.)

1. The complexity of the median chain is known to be not too big,
   $n^{4/3}$, by virtue of what tamal dey showed a while a ago
   (goes by the name "K-set bound"). i wanted to give this as a
   hint but realized it is not easy to see a connection from this
   to a sub-quad algorithm.

2. Let's sweep from left to right; we keep the current set of
   topmost n/2 lines in one data structure and the current set of
   bottommost n/2 lines in another. Lines will move from one
   structure to the other as the sweep proceeds. Each
   datastructure supports the following operations.

   In the bottom datastructure, we need to implement findNextMax
   as we sweep, delete a line, and insert a line into the top
   datastructure

   Likewise for the top.

3. The total number of deletes/inserts is bounded by $n^{4/3}$ by 1
   above because each such operation contributes a feature to the
   median chain. The number of findNextMaxes needs to be bound.

4. The number of times the max changes in the bottom
   datastructure is $n^{4/3}$ for reasons similar to 1. The main
   question is how long does each findNextMax take. Several sweep
   events may be needed before the max changes.

5. TO make findNextMax efficient, each datastructure is implemented recursively.

Split the set of lines in that datastructure into 2 halves; to
findNextMax, findNextMax in both halves and take max of the 2.
insert/delete just do so making sure the halves are roughly
balanced. In each half with say x lines, the max changes at
most $x^{4/3}$ times plus the number of insertions/deletions,
which in turn add up to $n^{4/3}$ over both halves.

6. This gives a recursion that comes to $O(n^{4/3})$ with some log
   factors...

===
=======
===
=======
          IOI Training Camp, Lecture Notes, Day 10

                     Tue 01 Jul 2008
===
=======
Session 1, 09:00--10:30
===
=======
Boundary (IOI 2003)

Farmer Don watches the fence that surrounds his N meter by N
meter square, flat field (2 <= N <= 500,000). One fence corner is
at the origin (0, 0) and the opposite corner is at (N,N); the
sides of Farmer Don's fence are parallel to the X and Y axes.

Fence posts appear at all four corners and also at every meter
along each side of the fence, for a total of 4N fence posts. The
fence posts are vertical and are considered to have no
radius. Farmer Don wants to determine how many of his fence posts
he can watch when he stands at a given location within his fence.

Farmer Don's field contains R (1 <= R <= 30,000) huge rocks that
obscure his view of some fence posts, as he is not tall enough to
look over any of these rocks. The base of each rock is a convex
polygon with nonzero area whose vertices are at integer
coordinates. Each rock is described by at most 20 vertices. The

rocks stand completely vertical. Rocks do not overlap, do not touch other rocks, and do not touch Farmer Don or the fence. Farmer Don does not touch the fence, does not stand within a rock, and does not stand on a rock.

Given the size of Farmer Don's fence, the locations and shapes of the rocks within it, and the location where Farmer Don stands, compute the number of fence posts that Farmer Don can see. If a vertex of a rock lines up perfectly with a fence post from Farmer Don's location, he is not able to see that fence post.

Solution:

Fix an interior grid point (X,Y). For each rock, we need to compute the segment of the fence that is blocked by the rock. For this, we have to compute the tangent points on the polygon. It is not difficult to see that the tangent must pass through a vertex of the polygon. There are K vertices. If we draw a line through each of these K vertices from (X,Y) and sort the slopes, the lines at the two extremes will be the tangents.

We number the trees 1..4N clockwise starting from the top left corner. For each rock r, we extend the tangents from (X,Y) to r to the boundary to identify a segment [L_r,R_r] of fence posts hidden by rock r. This takes time R (20 log 20) to process all the rocks. One minor point: if [L_r,R_r] include the top left corner --- that is, it starts before 4N and wraps around beyond 1 --- it is better to split it as two intervals [L_r,4N] and [1,R_r] to make all later calculations easier.

Now, for each fence post, we have to determine if it is hidden by some segment. If we do this naively, we will have to examine all R segments for each fence post, making the complexity NR.

We can do much better. We sort the R segments by the starting point. We then start walking from fence post 1. We maintain a count that keeps track of how many rocks are hiding the current fence post. Each time we enter a hidden segment, we increment the count. Each time we exit a hidden segment, we decrement the count. The current fence post is visible from (X,Y) precisely if the count is 0 when we reach that post.

It takes R log R time to sort the segments and then time N to

traverse the boundary and compute the number of visible fence posts.

===
=======
Frogs (IOI 2002)

A grid with plants at each point. Frogs jump across in straight lines entering the grid from one side and leaving the grid from the other side, landing at grid points. Each frog jumps with a fixed periodicity and flattens the plant that it lands on.

For instance, here is a 7 x 7 grid with four frog paths. Here "o" represents a good plant and "x" is a plant flattened by a frog. There are two horizontal paths in rows 3 and 7, one vertical path in column 4 and one diagonal path going through points (5,2),(4,4),(3,6). Note that a frog need not start its path at the edge of the grid.

```
          ^
          |
      o o o o o o o
      o o o x o o o
 <-- x o x o x x x ---->
      o o o x o o o
      o x o o o o o
      o o o x o o o
 <-- x x x x x x x ---->
          |
          v
```

In this task, you are given the set of grid points where plants have been flattened. From this, you have to compute the longest sequence of plants that could have been flattened on by a single frog. It is sufficient to find sequences of length 3 or more.

There are N flattened plants on an N x N grid, where N is up to 5000. Memory limit is 64 MB.

Solution

Brute force:

For each pair of frog points, extend it to a line and see if it
is a valid frog path. Keep track of the longest one.

How do we extend the line formed by two frog points to check if
it is a frog path? If we maintain the grid as an 0/1 array and
check whether the next point in the line is a 0 or 1. This uses
up about 25MB of space.

Testing each point is a constant. There could be upto N points
on this line, so this takes N steps per pair of points, or
$(N^2)*N = N^3$ time overall.

How can we improve this?

When we extend a pair to a line and check that it is a frog path,
we can mark all the following pairs of points as having been
checked. For instance, if we are checking (x,y) and $(x+d,y+d')$.
Then, we will examine $(x+2d,y+2d')$, $(x+3d,y+3d')$ .... We then
need not try to extend the pairs $\{(x+d,y+d'),(x+2d,y+2d')\}$,
$\{(x+2d,y+2d'),(x+3d,y+3d')\}$ to lines.

We still have to examine N^2 pairs of points, but for some pairs,
we have to do no work extending them. How do we measure the
complexity taking into account this reduction in work?

For the first pair of points $\{(x,y),(x+d,y+d')\}$, suppose we cover
M additional points when extending it to a path. This marks M-1
new pairs where no work has to be done. Thus, overall, we take M
steps to cover these M pairs. Overall, this averages to 1 step
per pair, so the algorithm takes only N^2 steps.

To record whether a pair of points has already been checked, we
need a second boolean array of size N x N (here the entry (i,j)
does not correspond to the coordinate of a point on the field but
rather to a pair of points (x_i,y_i) and (x_j,y_j) that have been
flattened by frogs). This takes another 25M of memory, so we are
within the overall limit.

How do we look at a pair of flattened points (x_i,y_i) and (x_j,y_j)
and determine the indices i and j to look up in the second array?
Sort the points, first by x and they by y. Find the index of a

point by binary search. This adds a log N factor to each
iteration, making it N^2 log N overall.

How do we eliminate the log N factor?

Consider the sorted array of plant positions.   Notice that a
frog path corresponds to a maximal arithmetic progression in this
array.

How do we efficiently find arithmetic progressions in this sorted
array? Construct a graph whose vertices are pairs (I,J) where I
and J are plant positions in the array. For each triple <I,J,K>
of consecutive plants on a line (i.e., distance+slope(I,J) =
distance+slope(J,K)), add an edge from vertex (I,J) to vertex
(J,K). In the resulting graph, the degree of each vertex is at
most 2 (each pair of plants on a line can be connected to at most
one neighbour on either side), we have a graph with N^2 vertices
and N^2 edges. Observe that each path in the grid is a connected
component in this graph. So, run DFS repeatedly to identify the
connected components and report the largest component. One minor
point is that we need frog paths to be maximal arithmetic
progressions within the grid, so for each component in the graph,
we need to check that if we extend the minimum and maximum pairs
in that component, we will go outside the grid.

To construct this graph, we need to compute equally spaced
triples <I,J,K>. Naively, this takes time N^2 log N --- for each
value I and J, we know the K we need, so we use binary search to
check if it exists.

We can do it in time N^2 as follows. For each value of I, start
two pointers J and K to the right of I. For a fixed value of J,
move K forward till you find a point equally spaced. If you
overshoot, stop the K pointer and move J forward till it is
midway between I and K. In this manner, move J and K right in
one scan. Whenever you find an equally spaced triple <I,J,K>,
update the graph to add an edge between (I,J) and (J,K).

===
=======
XOR (IOI 2002)

A N x N grid of pixels (can be set to white or black). At each step, specify we can make a call XOR(x1,y1,x2,y2). This selects a rectangle of pixesl with top left corner (x1,y1) and bottom right corner (x2,y2) and flips all the values in this rectangle (negation is the same as XOR with 1/black, hence the name).

For example (. denotes white, X denotes black)

```
  1 2 3 4 5 6 7 8 9              1 2 3 4 5 6 7 8 9
1 . . . . . . . . .            1 . . . . . . . . .
2 . . X X . . . . .              2 . . . . . . . . .
3 . X . . X . . . .              3 . X X X X . . . .
4 . X . . X . . . . XOR(2,3,5,4)  4 . X X X X . . . .
5 . . X X . . . . . ===========>  5 . . . . . . . . .
6 . X . . X . . . .              6 . X . . X . . . .
7 . X . . X . . . .              7 . X . . X . . . .
8 . X . . X . . . .              8 . X . . X . . . .
9 . . X X . . . . .              9 . . X X . . . . .
```

Given a final configuration, reconstruct the minimum number of XOR calls needed to reach it. (This is equivalent to asking whether we can run this sequence of XOR calls in reverse order from the final configuration and make the pixel grid entirely white.)

The instance above can be solved in 3 calls to XOR.

```
  1 2 3 4 5 6 7 8 9              1 2 3 4 5 6 7 8 9
1 . . . . . . . . .            1 . . . . . . . . .
2 . . X X . . . . .              2 . . . . . . . . .
3 . X . . X . . . . XOR(2,3,9,4)  3 . X X X X . . . . 2 moves
4 . X . . X . . . . =========>    4 . X X X X . . . . ==>
5 . . X X . . . . .              5 . . . . . . . . .
6 . X . . X . . . .              6 . X X X X . . . .
7 . X . . X . . . .              7 . X X X X . . . .
8 . X . . X . . . .              8 . X X X X . . . .
9 . . X X . . . . .              9 . . . . . . . . .
```

Solution:

Finding the optimal number of XOR calls exactly is

computationally very hard. Instead, we will find a strategy for approximating the optimal number of moves within a factor of 2.

For this, we identify a "measure" that we try to decrease with each XOR call. Look at the corners between pixels, where pixels meet. Each such point has four neighbours. A "corner" point is one that has odd parity (3 black, 1 white or vice versa). On the boundary, a corner point is one that has one black neighbour and one white neighbour.

Examples of corner points:

```
-----    -----    -----    -----    ---
|X|X|    | |X|    | | |    | |X|    |X|
-----    -----    -----    -----     --- boundary
| |X|    | | |    | |X|    |X|X|    | |
-----    -----    -----    -----    ---
```

Even if we have a single black point, we have four corners around it:

```
-------
| | | |
-------
| |X| |
-------
| | | |
-------
```

Clearly, the final all white grid has no corners.

When we select a rectangle to invert, only the points at the four vertices of the rectangle change parity. All internal and boundary points retain their parity. Therefore, an optimal sequence of XORs can at most remove 4 corners in each step.

If we can find a strategy to identify a rectangle with at least three corners in each step, we can reduce the number of corners overall by at least 2 at each step (this XOR will remove the three corners we have found and possibly add one back at the fourth vertex of the rectangle). In this way, we can systematically remove 2 corners at a time, achieving a sequence that is within a factor of 2 of the optima.

Claim: If there are any black pixels at all, we can always
identify a rectangle with at least three "corners".

Scan rows from the top and search for the first black pixel. The
top left point of this pixel has odd parity and is a corner. Go
right on this row to the rightmost black pixel (possibly the same
pixel). The top right point of this pixel has odd parity and is
a corner.

To find the third corner, from the first pixel, go down the
column till the first segment of black pixels ends. If this
is not a corner we must have a new segment of black pixels
starting diagonally opposite, as shown below:

```
     X
   X
   X
   X
   X
```

We continue down this column. The segment can alternate back and
forth a finite number of times

```
     X
   X
   X
   X
   X
      X
   X
   X
   X
   X
```

but must eventually stop, at least at the boundary, with a
corner.

We apply XOR to the rectangle defined by these three corners.
(The fourth corner pixel possibly does not define a "corner"
point).

1 2 3 4 5 6 7 8 9

```
1 . . . . . . . . . .
2 . . X -----> X . .
3 . X | . . . . . .
4 . X | . . . . . .
5 . . | . . . . . .
6 . X | . . . . . .
7 . X | . . . . . .
8 . X | . . . . . .
9 . . X . . . . . .
```

As we observed earlier, if we flip a rectangle with at least
three odd corners, the three odd corners become even (and,
perhaps, the fourth will go from even to odd). So, we are
guaranteed to reduce the number of odd corners by at least 2.

We are looking for a final figure with zero odd corners, so
eventually this strategy will solve the problem.

The optimal solution (whatever it may be) can at best remove 4
odd corners with each move. So, this strategy will take at most
twice the optimal, no matter what the optimal may be.

==
=======

Utopia (IOI 2002)

Number the four quadrants on the coordinate plane as follows

```
            |
     2      |    1
            |
  -------------+--------------
            |
     3      |    4
            |
```

You are given a sequence over {1,2,3,4}, e.g. 1244112431. This specifies an order in which the quadrants are to be visited --- you have to first move to quadrant 1, then quadrant 2, then quadrant 4, then stay in quadrant 4, ...

This tour consists of N stops, q1q2...qN. You are then provided with 2N distinct postive integers d1,d2,...,d2N. (N is in the range [1..10000].)

The tour starts at (0,0). We now have to group the 2N numbers into N pairs (x1,y1), (x2,y2), ..., (xN,yN). We can insert negative signs, if required, before each xi or yi.

Intepret these pairs as displacements. This constructs a tour as follows.

```
    Point              Current Quadrant

p0  (0,0)
p1  (x1,y1)                v1
p2  (x1+x2,y1+y2)           v2
p3  (x1+..+x3,y1+..+y3)     v3
.
.
.
pN  (x1+..+xN,y1+..+yN)      vN
```

The goal is to make the sequence of quadrants visited v1v2...vN equal to the original sequence q1q2...qN.

For instance, suppose the desired trajectory is 4 1 2 1 and there are 8 numbers 7, 5, 6, 1, 3, 2, 4, 8, here is a solution:

  (+7,-1), (-5,+2), (-4,+3), (+8,+6)

Simplified version:

We work along one axis, divided into two regions + or -.

```
        -             +
------------------|------------------
            0
```

We have a {+,-} sequence of length N that describes a trajectory. We are given N numbers. We begin at the origin. We have to fix an ordering of the N numbers and insert signs to obtain a sequence of N displacements that traces out the required trajectory.

E.g. Trajectory is   + + - +
    Values are   4 5 2 8

If we assign the signs as +4, -5, -2, +8, we can trace out the required trajectory by the sequence +4 -2 -5 +8.

Suppose we have a sorted sequence x1, x2, ..., xN in which we insert + and - signs alternately (we may start with + or - for x1).

Suppose we pick a subsequence xi,...,xj. Observe that the sign of sum(xi,...,xj) will always be the same as the sign of xj.

Pair xj with xj-1 etc. Each pair has the same sign as xj and they add up. If the sequence is of odd length, you are left with an extra unpaired element on the left, but this is of the same sign.

Now, observe the following. Given sum(xi,...,xj):

- if we add xj+1 (the next larger number with an opposite sign to
  xj), the sign of the sum (xi,...,xj+1) flips

- if we add xi-1 (the next smaller number with an opposite sign
  to xi), the sign of the sum (xi-1,...,xj) is the same as the
  sign of sum of (xi,...,xj)

We can thus use the sorted sequence with alternating signs provided we know

- how to assign the alternating signs
- where in the sorted sequence we start

Wherever we start, we move right each time we want to flip the sign and move left each time we retain the sign. So, we can determine the starting position by counting, in the given tour, the number of sign changes.

Having fixed the position that we start, the sign of this element
is determined by the sign of the first element of the sequence.
We then propagate alternating signs from this point.

In our earlier example:

  + + - +

The sorted sequence is 2 4 5 8. There is one move that does not
change sign, so we start at 4. The first side we have to visit
is +, so the sign of 4 is +. So, we work with the signed
sequence -2,+4,-5,+8. The order in which we choose the
displacements is +4, -2, -5, +8 (the order of points visited is
+4, +2, -3, +5).

Suppose instead the desired tour was

  + + - -

We now have two moves that do not change sign, so we start with 5
and make it +5, giving the sequence +2, -4, +5, -8. The moves
are now +5, -4, -8, +2 (points visited are +5, +1, -7, -5).

Observe that the solution is always feasible, for any choice of
traversal and displacements!

Back to 2 dimensional utopia

We can represent each quadrant in terms of the sign of the x and
y coordinates. Thus, 1 = (+,+), 2 = (-,+), 3 = (-,-), 4 =
(+,-). We can then split the sequence of quadrants into two
signed sequences, one for x and one for y. Divide the initial 2n
numbers into two sets of n, one for x and one for y and solve the
one dimensional problems for x and y independently.

===
=======
Reverse (IOI 2003)

We are given an array A with 9 elements A[1],A[2],...,A[9]. Each position in the array can store an integer between 0 and 255.

We are permitted the following operations:

0. Initialize each of the 9 elements to any legal value.

1. A[j] := A[i] + 1   /* The value of A[i] is unchanged. However
                we can have j == i, so we can also
                increment A[i] in place. */

2. Print A[i]

Given K, we have to write a "program" consisting of a single initialization step followed by a sequence of instructions of types 1 and 2 that prints out all values K, K-1, ..., 0 in descending order.

For instance, if K = 2, we can do the following:

1. Initialize array to {0,2,0,0,0,0,0,0,0}

2. Print A[2]   /* 2 */

3 A[2] = A[1] + 1

4. Print A[2]   /* 1 */

5. Print A[3]   /* 0 */

The cost of such a program is the longest sequence of consecutive assignments between any pair of adjacent print statements. Thus, for the program above, the cost is 1.

The task is to write a program of minimum cost for each K.

This is an output only task --- you are given 16 values of K in advance and only have to submit the programs for each of these values. In other words, you can construct each program by hand, without writing a program to generate these programs.

The upper bound on the cost of a valid Reverse program is 131. Solutions get 20% for correctness and 80% for optimality.

Simple solution

Load positions 1 to 7 with K/8, 2K/8, ..., 7K/8. Use position 8
for scratch work and keep position 9 as 0. (Note that we can not
generate a 0, so we have to preserve one value as 0).

Now, initially increment 7K/8 K/8 times into position 8 and print
K. Once we generate numbers upto 7K/8, we switch to starting
from 6K/8. Thus, the cost is at most K/8.

More sophisticated

Observe that once we pay the maximum cost at some point in the
program, we may as well uniformly take that many steps between
each pair of prints. Thus, we can organize the values in such a
way that we do auxiliary work "on the side" whenever the next
value is ready to print before we have reached our upper bound.

For instance, start with

K K-2 K-5 0

Assume we can manage with cost 1.

```
  Print A[1]           /* K  */
  A[1] = A[2] + 1
  Print A[1]           /* K-1 */
  A[1] = A[3] + 1        /* K-2 is ready to print, do some
                  auxiliary work */
  Print A[2]           /* K-2 */
  A[1] = A[1] + 1
  Print A[1]           /* K-3 */
  A[1] = A[3] + 1
  Print A[1]           /* K-4 */

  ...  /* K-5 is ready so do some auxiliary work */
```

In this way, we can fix a sequence of values assuming we have
cost 1 and see how far we can go.

If we get stuck, we revise our target cost to 2 and recompute a
new set of optimum initial values.

It turns out that K=255 can be programmed with cost 4!

Sorting

We want to sort an array with N elements.

Naive algorithms such as bubblesort and insertion sort take N^2
operations to sort the array. Better sorting algorithms work in
time proportional to N log N.

Is this improvement worth it?

Consider the problem of searching for a value in an array of size
10000. We can either sort the array and then use binary search
(that takes time log N) or do an exhaustive search.

Exhaustive search takes time N, in general, since we may have to
scan the entire array. For our example, we can thus search for a
single value in 10000 steps.

Binary search takes time log N (all logs are to the base 2).
Recall that $2^{10} = 1032$ so log 1000 is approx 10. Since 10000 =
1000 * 10, log 10000 = log 1000 + log 10 = 10 + 4 = 14.

If we are searching for a single value, we can use exhaustive
search and do the job in 10000 steps. If we
sort the array using an N^2 sorting algorithm and then use binary
search, we spend 10000*10000 = 10^8 steps sorting it, followed by
14 steps to search.

If we repeat this search 10000 times, we would spend 10000*10000
operations doing exhaustive search and 10000*10000 + 14*10000 =
10014*10000 operations doing sorting plus binary search. For any
value above 10014, sorting will start to make itself

effective---if we check when 10000*M = 10000*10000 + 14*M, we get
M a little above 10014.

On the other hand, suppose we use an N log N sorting algorithm.
Then, the sort phase takes time 14*10000. So, to see when
sorting+binary search beats exhaustive search, we need to check
when 10000*M = 14*10000 + 14*M, which is approximately when M =
14. Thus, if we do even 15 searches, it helps to sort.

What if the array grows from 10000 to 100000? Since log 100000
is about 17, the equations we have are

  N^2 sorting: 100000*M = 100000*100000 + 17*M
  N log N sorting: 100000*M = 100000*17 + 17*M

Thus, we need to about 100017 exhaustive searches to justify
justify an N^2 sorting algorithm, but only 17 searches to justify
an N log N sorting algorithm.

More importantly, if we use an N log N sorting algorithm, for
values much less than the size of the array, we can use sorting
to significantly speed up the search process.

The reason why bubblesort etc are inefficient is that they
duplicate comparisons. One way to avoid this is to break up the
array into non overlapping segments, sort these and then combine
these portions efficiently.

For instance, suppose we have a box with slips of paper with
numbers between 100 and 300 to be sorted. We could first
separate the slips into two bunches: those with values below 200
and those with values above 200. We can sort these bunches
separately. Since all the values in the second bunch dominate
those in the first bunch, we can combine them easily into a
single sorted bunch.

Merge sort

In general, there may not be a simple way to determine how to
split the array into two halves based on the values. Instead, we
can just split the array arbitrarily at the midpoint. We sort
each half separately. We then have to combine these two halves,

but we are not guaranteed that one half is uniformly below the second half.

Let N = 2M be the size of the array. Suppose, after sorting the two parts, we have the array in the form

   i_1 i_2 ... i_M j_1 j_2 ... j_M

where i_1 i_2 ... i_M and j_1 j_2 ... j_M are each sorted half arrays.

We can then proceed as follows. Examine i_1 and j_1 and extract the minimum value into the first position of a new array. Suppose i_1 is selected. In the next step, compare i_2 and j_1 and extract the minimum value. Proceed in this fashion till we exhaust both halves. (If we find that the elements are equal, we take the value from the left.)

In general, this merge operation takes time N. Let T(N) denote the time taken to sort an array of N elements. This approach requires us to first sort two arrays of N/2 elements and then combine them using N steps. So

   $T(N) = 2*T(N/2) + N$

We recursively solve the two halves using the same procedure. Thus, the same equation applies to T(N/2), so we can unravel this equation as

   $T(N) = 2*(2*T(N/4) + N/2) + N = 4*T(N/4) + 2N$
       $= 4*(2*T(N/8) + N/4) + 2N = 8*T(N/8) + 3N = ...$

After expanding this equation K times, we get

   $T(N) = 2^K * T(N/2^K) + KN$

Eventually, when K = log_2 N, we get

   $T(N) = 2^{(\log_2 N)} * T(1) + (\log_2 N)*N$

T(1), the time to sort a 1 element array, is 1 so we can simplify the equation above to get

$T(N) = N + N*(\log\_2 N)$

We ignore the smaller term and get T(N) to be proportional to
N*(log_2 N), which is considerably better than N^2. (In computer
science, 2 is the default base for logarithms, so we shall
henceforth write log N to mean log_2 N).

The problem with this algorithm is that we need an extra array of
size N to merge the two halves. Can we avoid this?

Quicksort

Quicksort provides a method to do this that works well on the
average.

1. Call A[0], the first element of the given array A the PIVOT
   element or the SPLITTER.

2. We regroup A[1] to A[N-1] into two parts. The first part
   consists of elements less than the splitter A[0] while the
   second part consists of elements greater than A[0]. (If there
   are duplicate entries in the array, we may have elements equal
   to A[0]. We can place these, for instance, in the first
   partition.)

3. After rearranging the elements, we move the splitter element in
   between the two partitions and recursively sort the lower and upper
   parts of the array (leaving out the splitter).

We will shortly describe how to do this rearrangement
efficiently. Before that, we observe that this procedure will
not necessarily give us subproblems of size N/2 as we had assumed
in the earlier analysis. The sizes of the two partitions depend
on the value of A[0]. In the worst case, A[0] may be the
smallest or largest value in A, resulting in one of the two
partitions being empty. However, such extreme cases are rare and
it can be proved mathematically that on an average Quicksort runs
in time proportional to N * log N. More importantly, Quicksort
works very well in practice and it is usually the algorithm used
to implement built-in sorting routines.

To achieve the rearrangement, we scan the array from A[1] to A[n-1]. At an intermediate point in the scan, we are examining A[i]. At this point, we should have partitioned A[1] to A[i-1] into two groups, those smaller than A[0] and those bigger than A[0]. We thus have two indices into the array, b and i. The index b specifies the boundary between the lower and upper partitions within A[1] to A[i-1]. We can fix b to be either the last position in the lower partition or the first position in the upper partition. Suppose we choose the latter.

We then get an invariant as follows:

After each iteration of the loop where we scan through A,
   i marks the first element of the part of the array that is yet to
     be scanned
   b marks the beginning of the upper partition within A[1] to A[i-1]

Thus, at an intermediate point, the array looks like the following.

```
  0 1       b     i      n-1
  ---------------------------------------
 | | <= A[0]  | > A[0]  | Yet to scan  |
  ---------------------------------------
```

When we scan A[i], we check whether it is smaller than A[0]. If so, it should move to the smaller partition, so we swap it with the entry at A[b] and increment both b and i. If A[i] is not smaller than A[0], it is already in the correct partition so we leave b unchanged and just increment i.

```
  i = 1; b = 1;
  while (i < n){
    if (A[i] <= A[0]){
      swap(A[b],A[i]);
      b++;
    }
    i++;
  }
```

Of course, at the end we should also insert the splitter between the two partitions, so we can add, after the loop;

swap(A[0],A[b-1]); /* Insert splitter before upper partition */

We then recursively sort A[0]..A[b-2] and A[b]..A[n-1]. As we
mentioned earlier, in the worst case, one of these two intervals will
have length n-1 while the other is empty, but on an average the array
will split into approximately equal parts.

In general, quicksort will be a recursive function which will be
passed the array and the left and right endpoints of the interval
to be sorted. We use the convention that the left endpoint is
the first index in the interval while the right endpoint is one
position beyond the last index in the interval. Thus, the
initial call would be to quicksort(A,0,n), where n is the size of
A. In general, the function is

```
  int quicksort(int *A, int l, int r){

    if (r - l <= 1)){
      return 0;    /* Trivial array, do nothing */
    }

    /* Rearrange with respect to splitter, a[l] */

    b = l+1;
    for (i = l+1; i < r; i++){
      if (A[i] <= A[l]){
        swap(A[b],A[i]);
        b++;
      }
    }
    swap(A[l],A[b-1]);  /* Insert splitter before upper partition */

    /* Recursive calls */

    quicksort(a,l,b-1);
    quicksort(a,b,r);
}
```

Quicksort in real life

If we choose the pivot element to be first (or last) element in

the array, the worst case input for quicksort is when the array
is already sorted. We argued that this will rarely happen and
that in the average case, quicksort works well in time n log n.

In certain practical situations, the worst case can and does
happen with unexpected frequency. Consider an application like a
spreadsheet where you can display and manipulate a table of data.
One option typically provided to the user is to select a column
and sort the entire table based on the values in that column. It
is often the case that the user first sorts the column in the
wrong order (ascending or descending), realizes the mistake and
then repeats the sort in the opposite direction. If the column
sorting algorithm is quicksort, the second sort provides a worst
case input!

One solution is to randomize the choice of pivot. In Unix, you
can generate random numbers as follows:

```
 #include <stdlib.h>

 int seed;
 ...
 srandom(seed);  /* Initialize the random number generator*/


 ...

 j = random();   /* Returns a integer random number */
```

If we initialize srandom() with the same value, the subsequent
sequence of random numbers will be the same. In IOI, if you use
randomization in your program, it is required that your random
number generator always yields the same sequence for any input.
Thus, you should use a fixed value as a seed, not something
"random" like the current system time etc.

To generate a random number in the range 1..k, we can write

```
 j = random()%k;
```

For quicksort, we need to choose a random pivot element between
positions l and r-1. We can do this by writing:

```
 pivot = l+ random()%(r-l);
```

swap(A[l],A[pivot]);

The second statement moves the pivot to the beginning of the
array. We can then use the earlier implementation of quicksort
that assumes that the pivot is at the beginning of the array.

==
=======

Stable sorting

An additional property that is desirable in a sorting algorithm
is stability. This says that two values that are equal in the
original array are never swapped in the process of sorting.

This is important if we sort data multiple times on different
criteria. Suppose we have an array in which values are pair of
the form (name, marks). We want the final output to have the
arrays arranged in ascending order of marks. Since there may be
multiple students with the same marks, we want to sort the
students with the same marks by alphabetical order of name.

If we have a stable sorting algorithm, we can do this as follows:

1. First sort the whole array by name
2. Then sort the whole array by marks

The second sort does not disturb the order of students with equal
marks, so all students with equal marks retain their alphabetical
order.

Mergesort is a stable sort (we ensured this by saying that we
pick the element from the left when merging if the two values
being compared are equal.)

Quicksort, as described above, is not stable. (Why?)