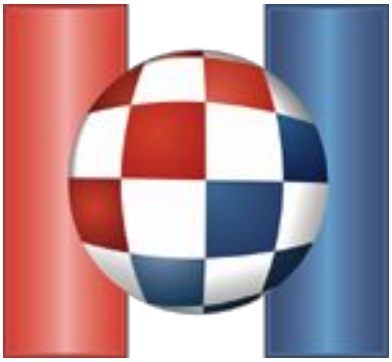
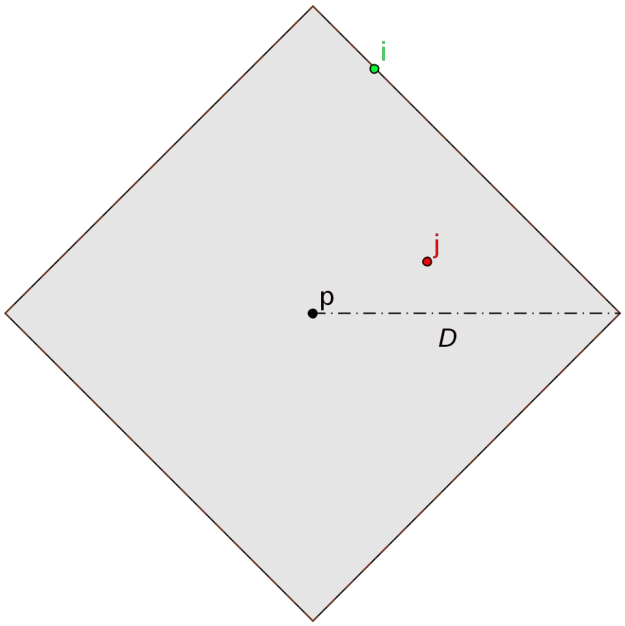


FINAL EXAM #2
April 16th 2015
solutions

Task DOSTAVA	**Author: Adrian Satja Kurdija**

Let us first solve this problem for a single point *p* and corresponding distance *D*. The points that with distance *D* (according to *Manhattan* metrics) between them and the point *p* form a square as depicted:



It is sufficient to choose a point (for example, point *i*) on the edge of this square. Moreover, we mustn't choose a point located inside the square (for example, point *j*).

In case of multiple points p_i, for each one, we observe the square for its distance D_i and there has to exist at least one point on the edge of each square. As before, not a single point from the solution mustn't be located inside one of the squares.

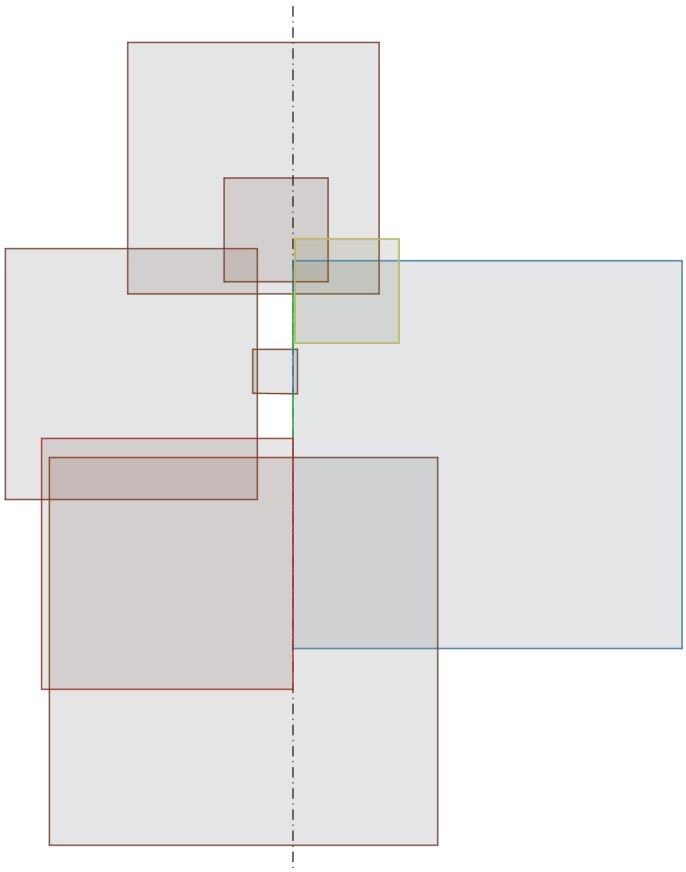
So the idea is to choose (at least one) point on each square.

Initially, we will rotate the plane for 45 degrees, so the sides of the square align with the coordinate axes. We can achieve this by the following transformation of point (x,y) to (x', y'):
x' = x + y
y' = x - y
The inverse transformation is of course:
x = (x' + y') / 2
y = (x' - y') / 2
We need to keep in mind here that the points with different coordinate parity after the inverse transformation are not integers, so we will take into consideration only the points with the same parity in the transformed plane.

The plan is to try and find a point with integer coordinates on each of the sides of each square.

We will look at the procedure of finding the points on the vertical sides of the square, the horizontal ones reduce to the same thing when rotating the plane for 90 degrees.

We will use the *sweep-line* algorithm where the line moves from left to right and at all times remembers which line segments cross the squares in an appropriate structure. When we come across a square side, the structure will find (if there exists one) a point on it.



The image depicts the moment when the algorithm is processing the left side of the blue square. In the structure are stored the y-intervals of the brown rectangles. The y-interval of the red square (which right side is on the same coordinate as the left side of the blue square) was just removed from the structure, and the y-interval of the yellow rectangle (which left side matches with the left side of the blue square) still hasn't been added to the structure, nor has the y-interval of the blue rectangle. Two green segments depict the area where the structure is going to search for points. We see that it makes sense, for an x-coordinate, to first remove all the rectangles that end on it, then ask the structure for the sides of all the rectangles that begin or end on it, and only then add all the rectangles that begin on it. We can do this in multiple ways, here the *events* are defined so that each is processed exactly once.

For simplicity's sake, we will use three types of *events*:

1) square ending - remove the square's y-interval from the structure
2) square side - we ask the structure for a point with integer coordinates in the y-interval
3) square beginning - add the square's y-interval in the structure

For each square, we will create four *events*: for the left side *events* of type 3 and 2, for the right *events* of type 1 and 2. The sorting criteria is primarily by the *x* coordinate, then by the type number. We will achieve that in the moment when we're searching for points on the sides on the current *x* coordinate, the structure contains y-intervals of those squares that are only being strictly crossed by the *sweep* line.

The data structure should support the following operations:
increase_interval(lo, hi) => increase counters in the interval [lo, hi]
decrease_interval(lo, hi) => decrease (**already increased**) counters in the interval [lo, hi]
give_zero(lo, hi, p) => return the index of the parity *p* of the counter that contains a zero from the interval [lo, hi]

Two things should be mentioned. The operation decrease_interval will always decrease the y-interval of the square which the same interval was increased before (because the we perform the decrease on the right side of the square, and the increment then had to happen before, on the left side). This slightly simplifies the structure implementation. The parity is important for the operation give_zero in order to ask the structure for even indices on even *x* coordinates, and for odd indices on odd coordinates (this way we are sure that we observe only the integer coordinates).

We will use the *tournament* tree for the structure. In each node, we remember the counter for its interval, so the equivalent counter for an index is equal to the sum of the counters of nodes on the path from the root of the tree to the corresponding leaf. Additionally, we remember two indices for each node, one even and one odd, at which the value of the equivalent counter is equal to zero. If such indices don't exist for a node, we can set them to an "undefined" value. All such values is easy to maintain with aforementioned operations, and for implementation details consult the source codes of the official solutions.

In the end, we will notice that the coordinates in the task are quite large. This problem can be solved in two ways: 1) we can implement the structure using pointers, so that we "discover" nodes only when they are needed, or 2) we can make a compression of the coordinates, so that we take into account only the important coordinates.

Task NAFTA	**Author: Gustav Matula**

As a start, it is good to notice that an oil drill pumps oil from a pool if and only if it is inside the interval from the leftmost to the rightmost field of that pool. Now we can reformulate the task in the following way:
You are given a set of **N** intervals with boundaries from [1, **S**] and values from [0, 9*__RS__] (which isn't really important). We need to choose at maximum **K** numbers from [1, **S**] so that the sum of the values from the interval for which we chose at least one number is maximal. We need to note that the number of intervals **N** can be O(**RS**).

This task can be solved using dynamic programming.
Let us denote the maximal sum of values as *dp(i, k)* if we know that we've already chosen the number *i* and that we are left with *k* more numbers to choose. We choose to take the first next number *j* from [*i* + 1, **S**]. We are interested in the sum of values from the interval that we didn't have so far - more precisely, the sum of values from the interval that contain *j*, but not *i*. Let us denote that sum with *w(i, j)*. We have the following relation:
$$dp(i,\ k)\ =\ min\{w(i,\ j)\ +\ dp(j,\ k\ -\ 1),\ for\ j\ from\ [i\ +\ 1,\ S]\}$$

The complexity of calculating the value *dp* is O(**KS**^2), which was enough for the first two tasks, assuming that *w* is calculated fast enough (the algorithm that go through all intervals for each pair *(i, j)* are too slow). The value *w* can also be calculated using dynamic programming. Let us denote the sum of values from the interval that **begin** with *i* and **contain** *j* with *I(i, j)*. This can easily be calculated in the complexity O(**RS**) (for interval [l, r] we increase *I(l, j)* for the value from the interval for *j* from [l, r]).
Now for *w* the relation *w(i, j) = w(i + 1, j) + I(i + 1, j)* holds.
The total complexity of calculating *w* is O(**RS + S^**2).

We are left with fast computation of *dp*. For that we will use a classic technique. We define *p(i, k)* as the **minimal** *j* so that it holds *dp(i, k) = w(i, j) + dp(j, k - 1).*
Therefore, *p(i, k)* is the optimal transition from the state *(i, k)*.
It is possible to prove (see the end of the description for the informal proof) that it holds *p(i, k) <= p(i + 1, k)*, and it allows us to apply the *divide & conquer* paradigm in the following way.

Let us assume that we calculated *dp(i, k - 1)* for all *i*. We want to calculate *dp(i, k)* for all *i*. From the relation it is evident that the order in which we calculate this isn't important, so we will first calculate *dp(S / 2, k)* and at the same time we will find *p(S / 2, k)* (/ 2 denotes integer division here). The complexity of this step is O(**S**).

Then we will separately and recursively calculate *dp(i, k)* for *i < S / 2*, or for *i > S / 2*.
We need to keep in mind that for the left part of the recursion it holds *p(i, k) <= p(S / 2, k)*, and for the right part *p(i, k) >= p(S / 2, k)*, which means that in the next level in the left branch we will only visit the interval [1, S / 2 - 1], and in the right branch [S / 2 + 1, S] so the next level of recursion will again be of complexity O(**S**).

We can formalize this procedure so that we parametrize the recursion with four values: *lo, hi, plo, phi*. Then [*lo, hi*] denotes the interval of value *i* for which we want to calculate *dp(i, k)*, and [*plo, phi*] the interval in which the optimal transition for all *i* from [*lo, hi*] is located. The recursion will look like this:

```
solve(lo, hi, plo, phi):
      mid = (lo + hi) / 2
      calculate dp(mid, k) and p(mid, k)
      solve(lo, mid - 1, plo, p(mid, k))
      solve(mid + 1, hi, p(mid, k), phi)
```

The recursion will, because of breaking in halves, have O(*log* **S**) levels, and on each level we will make O(**S**) steps in total (because the adjacent intervals [*plo, phi*] overlap on each level only in the edge cases).

We need to apply this process to **K** (for each k to **K**), the total complexity is O(**KS** *log* **S**).

We are left with sketching the proof that *p(i, k) <= p(i + 1, k)*.
For *p(i, k) = i + 1* the claim is trivial.

Otherwise, let us assume the contrary, that *p(i, k) > p(i + 1, k)*.

From definition *p* it follows:
dp(i, k) = w(i, p(i, k)) + dp(p(i, k), k - 1) > w(i, p(i + 1, k)) + dp(p(i + 1, k), k - 1)
The equation is not possible because in that case by definition *p* it would hold *p(i, k) = p(i + 1, k)*

It also holds:
dp(i + 1, k) = w(i + 1, p(i + 1, k)) + dp(p(i + 1, k), k - 1) >= w(i + 1, p(i, k)) + dp(p(i, k), k - 1)

By summing up the two equations and cleaning up, we get:

w(i, p(i, k)) + w(i + 1, p(i + 1, k)) > w(i, p(i + 1, k)) + w(i + 1, p(i, k))

Using the relation for *w* it follows

$w(i, p(i, k)) = w(i + 1, p(i, k)) + I(i + 1, p(i, k))$, and
$w(i, p(i + 1, k)) = w(i + 1, p(i + 1, k)) + I(i + 1, p(i + 1, k))$, so by inserting in the equation and cleaning up we get:

$I(i + 1, p(i, k)) > I(i + 1, p(i + 1, k))$

Therefore, we got that the sum of intervals that begin with $i + 1$ and contain $p(i, j)$ is greater than the sum of intervals that begin with $i + 1$ and contain $p(i + 1, k)$, which is possible only if $p(i, k) < p(i + 1, k)$, therefore we have a contradiction.