École Polytechnique — CSE202 Design and Analysis of Algorithms

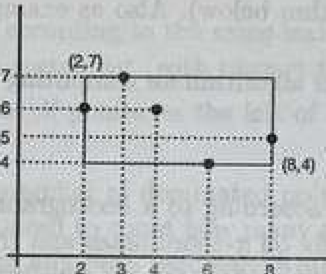
Midterm Exam: November 4th, 2021

1 Divide and Conquer : Playing with Points on a Plane

Guidelines. Download `points.py` and `test_points.py`. Write your solution in `points.py`. To test your code, run the file `test_points.py`.

1.1 Bounding Box

A very common problem in computer graphics is to approximate a complex shape with a bounding box. For a set S of $n \geq 1$ points in 2-dimensional space, the idea is to find the *smallest* rectangle R with sides parallel to the coordinate axes that contains all the points in S.



The drawing above gives an example of such a rectangle. The rectangle is defined by the coordinates of the top-left corner $(2,7)$ and bottom-right corner $(8,4)$. We assume that the set of points is given as a list of 2-size lists representing the x and y coordinates of a point, respectively. For the instance in the figure, the set of points is given as $[[2,6],[3,7],[4,6],[6,4],[8,5]]$. For a set of points of size 1 $[[x,y]]$, the rectangle is "empty" and the coordinates of the top-left and bottom-right corner are both (x,y).

The construction of R can be reduced to two instances of the problem of simultaneously finding the minimum and the maximum in a set of n numbers; namely, we need only do this for the x-coordinates of points in S and then for the y-coordinates of points in S.

Question 1 (2 points). Complete the function `min_max(A)` with a divide-and-conquer algorithm for returning both the minimum and the maximum element of the list `A` using no more than $3n/2$ comparisons. Justify its complexity by writing at least the recurrence relation, including the value for the first terms. (Write the answer as a comment in the code.)

```
In [1]: min_max([1,43,232,43,23,5,7,-1])
Out[1]: (-1, 232)
```

It is probably useful to define an intermediate function `min_max_aux(A,left,right)` that finds the minimum and maximum for the sub-array from index `left` to `right` (since, as expected, you are going to divide the array in two halves and compute min/max for each half recursively).

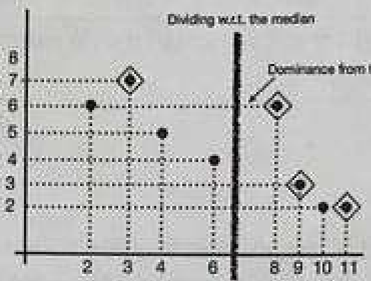
Question 2 (2 points). Complete the function `bounding_box(S)` that returns the smallest rectangle R that contains all the points in S. As indicated above, the function should return the coordinates of the top-left and bottom-right corners of the rectangle.

```
In [2]: bounding_box([[2,6],[3,7],[4,6],[6,4],[8,5]])
Out[2]: [[2, 7], [8, 4]]
```

1.2 Maxima Set

We consider the problem of finding a maxima set for a set S of n points in the plane. This problem is motivated from multi-objective optimization, where we are interested in optimizing choices that depend on multiple variables.

A point (x, y) *dominates* another point (x', y') when $x \geq x'$ and $y \geq y'$. A point $(x, y) \in S$ is a *maximum* point in S if there is no other point $(x', y') \in S$ such that (x', y') dominates (x, y). We are interested in the problem of computing the set of maximum points, called *maxima set*, for a set S.



In the figure above, the points surrounded by a diamond form the maxima set (the other explanations concern the divide-and-conquer algorithm below). Also as example, the point $(3, 7)$ dominates $(2, 6)$ but not $(4, 5)$.

One can define a divide-and-conquer algorithm for computing the maxima set as follows. The figure above gives more intuition. Thus,

— if $n \leq 1$, then return S

— let p be the *median* point in S according to a lexicographic ordering of the points in S, that is, where we order based primarily on x-coordinates and then by y-coordinates if there are ties. If the points have distinct x-coordinates, then we can imagine that we are dividing S using a vertical line as in the figure.

— we solve the maxima-set problem for the set of points on the left of this median and also for the points on the right.

— the maxima set of points on the right are also maxima points for S. But some of the maxima points for the left set might be dominated by a maximum point from the right (if a point a is before another point b in the lexicographic ordering, then a may or may not be dominated by b). It is sufficient to consider the point q that is the smallest among right set maxima points (w.r.t. the lexicographic ordering). For instance, in the figure, the point q is $(8, 6)$, and it dominates $(4, 5)$ and $(6, 4)$ which are maxima points for the left set.

So then we do a scan of the left set of maxima, removing any points that are dominated by q, until reaching the point where q's dominance extends. The union of remaining set of maxima from the left and the maxima set from the right is the set of maxima for S.

To simplify the computation of the median, we will assume that the set S is given as a list of 2-size lists as before, which is also ***sorted*** w.r.t. the lexicographic ordering mentioned above. Also, you can consider that the median of a set S of size n is the $\lfloor n/2 \rfloor$ smallest element.

Question 3 (3 points). Complete the function `maxima_set(S)` that implements this algorithm and returns the maxima set for S as a list of points. Derive its complexity by writing at least the recurrence relation. (Write the answer as a comment in the code.)

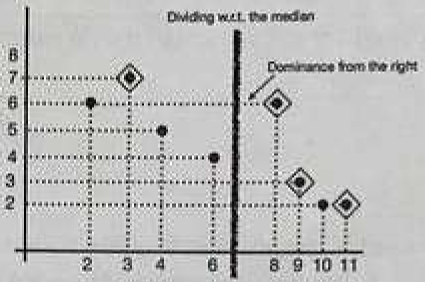
```
In [3]: A = [[2,6],[3,7],[4,5],[6,4],[8,6],[9,3],[10,2],[11,2]]
In [4]: maxima_set(A)
Out[4]: [[3, 7], [8, 6], [9, 3], [11, 2]]
```

The file `points.py` contains a function `lexicographic(p1,p2)` that checks whether `p1` is before `p2` in the lexicographic ordering described above. You can use it in your code.

1.2 Maxima Set

We consider the problem of finding a maxima set for a set S of n points in the plane. This problem is motivated from multi-objective optimization, where we are interested in optimizing choices that depend on multiple variables.

A point (x, y) *dominates* another point (x', y') when $x \geq x'$ and $y \geq y'$. A point $(x, y) \in S$ is a *maximum* point in S if there is no other point $(x', y') \in S$ such that (x', y') dominates (x, y). We are interested in the problem of computing the set of maximum points, called *maxima set*, for a set S.



In the figure above, the points surrounded by a diamond form the maxima set (the other explanations concern the divide-and-conquer algorithm below). Also as example, the point $(3, 7)$ dominates $(2, 6)$ but not $(4, 5)$.

One can define a divide-and-conquer algorithm for computing the maxima set as follows. The figure above gives more intuition. Thus,

- if $n \leq 1$, then return S
- let p be the *median* point in S according to a lexicographic ordering of the points in S, that is, where we order based primarily on x-coordinates and then by y-coordinates if there are ties. If the points have distinct x-coordinates, then we can imagine that we are dividing S using a vertical line as in the figure.
- we solve the maxima-set problem for the set of points on the left of this median and also for the points on the right.
- the maxima set of points on the right are also maxima points for S. But some of the maxima points for the left set might be dominated by a maximum point from the right (if a point a is before another point b in the lexicographic ordering, then a may or may not be dominated by b). It is sufficient to consider the point q that is the smallest among right set maxima points (w.r.t. the lexicographic ordering). For instance, in the figure, the point q is $(8, 6)$, and it dominates $(4, 5)$ and $(6, 4)$ which are maxima points for the left set.

  So then we do a scan of the left set of maxima, removing any points that are dominated by q, until reaching the point where q's dominance extends. The union of remaining set of maxima from the left and the maxima set from the right is the set of maxima for S.

To simplify the computation of the median, we will assume that the set S is given as a list of 2-size lists as before, which is also ***sorted*** w.r.t. the lexicographic ordering mentioned above. Also, you can consider that the median of a set S of size n is the $\lfloor n/2 \rfloor$ smallest element.

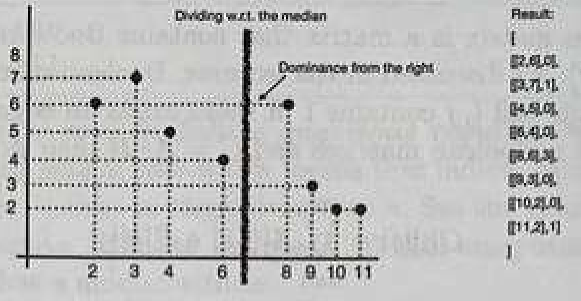
Question 3 (3 points). Complete the function `maxima_set(S)` that implements this algorithm and returns the maxima set for S as a list of points. Derive its complexity by writing at least the recurrence relation. (Write the answer as a comment in the code.)

```
In [3]: A = [[2,6],[3,7],[4,5],[6,4],[8,6],[9,3],[10,2],[11,2]]
In [4]: maxima_set(A)
Out[4]: [[3, 7], [8, 6], [9, 3], [11, 2]]
```

The file `points.py` contains a function `lexicographic(p1,p2)` that checks whether `p1` is before `p2` in the lexicographic ordering described above. You can use it in your code.

1.3 Dominance Counting

For a set S of points, we consider the problem of computing, for each point $a \in S$, the number of *other* points in S that are dominated by a. We assume that the result is computed as a list of 2-size lists $[a, n_a]$ where a is a point in S (it is itself a 2-size list containing the two coordinates) and n_a is the number of points dominated by a. The figure below provides an example of an expected result.



One can define a divide-and-conquer algorithm for counting dominated points as follows :

— if $n = 0$, then return the empty list, and if $n = 1$, then return $[S[0], 0]$ (the only point $S[0]$ in S dominates 0 other points).

— let p be the *median* point in S according to the same lexicographic ordering as above. As before, we assume that S is given as a sorted list (with respect to the lexicographic ordering).

— we solve the problem for the set of points on the left of this median and also for the points on the right.

— for the points on the left, the number of dominated points does not change in the final result. For the points on the right, we need to count how many points from the left they dominate. For this, you could use a brute-force approach (enumerate all points from the left, for each point on the right). But, this will not be optimal. Instead, you can sort the points on the left w.r.t. their y coordinate, and use binary search. For the final result, for each point on the left, we return the count from the recursive call, and for each point on the right, we return the count from the recursive call plus the additional count of points from the left that it dominates.

Question 4 (3 points). Complete the function `dominance_counting(S)` that implements this algorithm and returns the count as a list described above.

```
In [5]: dominance_counting(A)
Out[5]:
[[[6, 4], 0],
 [[4, 5], 0],
 [[2, 6], 0],
 [[3, 7], 1],
 [[9, 3], 0],
 [[8, 6], 3],
 [[10, 2], 0],
 [[11, 2], 1]]
```

The file `points.py` contains a function `sort_y(C)` that allows to sort the result of the recursive call with respect to the y coordinate (it assumes that `C` is constructed correctly as indicated above). This function does not return anything, you can just call it with some list and it will sort it.

2 Randomization : Witnessing Boolean Matrix Multiplication

Guidelines. Download `witness.py` and `test_witness.py`. Write your solution in `witness.py`. To test your code, run the file `test_witness.py`. The file `witness.py` contains a function `multiply_matrices(A,B)` that returns the product of two integer matrices A and B, and a function `inverse(A)` that returns $-A$ for a matrix A.

We recall that a *Boolean* matrix is a matrix that contains Boolean values 0 or 1 (interpreted as False and True, respectively). As discussed in the lectures, Boolean matrices can be used to represent graph adjacency matrices (the cell i, j contains 1 iff there exists an edge from vertex i to vertex j).

Let A and B be two $n \times n$ Boolean matrices and $C = A \cdot B$ their product. We recall that

$$C[i][j] = \bigvee_{k=1}^{n} A[i][k] \wedge B[k][j]$$

where $\vee$ is Boolean disjunction and $\wedge$ is Boolean conjunction. ***In the following, we assume that row or column indices range from 1 to n.***

A *witness* for $C[i][j]$ is an index $k \in \{1, \ldots, n\}$ such that $A[i][k] = B[k][j] = 1$. Remark that $C[i][j] = 1$ iff $C[i][j]$ has some witness k. A *Boolean product witness matrix* (BPWM) for C is an integer matrix W (containing integer values) such that each entry $W[i][j]$ contains a witness k for $C[i][j]$ if any, and it is 0, if there is no such witness. The values stored in the matrix W belong to the set $\{0, \ldots, n\}$ (the use of 0 to denote lack of a witness motivates the index range to start from 1 and not 0 as usual). Each entry $C[i][j]$ can have many witnesses and therefore, a BPWM is not unique.

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \quad B = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \quad C = A \cdot B = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \quad W = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}$$

For example, C is the product of the matrices A and B on the left (note that $C[1][2]$ is 1 and not 2 since the 0,1 values are Boolean values and the operations applied are Boolean disjunction and conjunction). The matrix W is a BPWM for C. The entry $W[1,1]$ is 2 because $A[1][2] = 1$ and $B[2][1] = 1$. Note that non-zero values in W correspond to non-zero values in C. Also, you may remark that $C[1][2]$ has two possible witnesses, the index 1 in W, but also the index 2.

A matrix will be represented in Python as a list of lists (rows), e.g., A is represented as $[[1, 1], [1, 0]]$.

The goal of this exercise is to solve the *BPWM problem* defined as follows :

Input : $n \times n$ Boolean matrices A and B
Output : A witness matrix W for $C = A \cdot B$

Question 5. [2 points] Complete the function `compute_witness_trivial(A,B)` to return a witness for $C = A \cdot B$ computed using a brute-force approach, i.e., search for a witness for each $C[i][j]$ by enumerating all indices $k \in \{1, \cdots, n\}$.

```
In [6]: A = [[1,1],[1,0]]
In [7]: B = [[0,1],[1,0]]
In [8]: compute_witness_trivial(A,B)
Out[8]: [[2, 1], [0, 1]]
```

What is the complexity of this algorithm ? (Write the answer as a comment in the code.)

Next, we describe a randomized algorithm for solving the BPWM problem.

Consider first the issue of finding a witness matrix when there is a *unique* witness for each entry $C[i][j]$. In this case, there is a simple reduction to *integer* matrix multiplication. Consider the matrix $\hat{A}$ defined by $\hat{A}[i][k] = k \cdot A[i][k]$ for each i, k (here, $\cdot$ is the standard multiplication of integers). If each entry $C[i][j]$ has a unique witness, then $W = \hat{A} \cdot B$ is a witness matrix W for C. Here, $\hat{A}$ and B are multiplied as integer matrices.

For instance,

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \quad \hat{A} = \begin{pmatrix} 1 & 2 \\ 1 & 0 \end{pmatrix} \quad B = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad W = \hat{A} \cdot B = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}$$

Question 6. [2 points] Complete the function `expose_column(A)` that returns the matrix $\hat{A}$ corresponding to the input matrix A.

```
In [9]: expose_column(A)
Out[9]: [[1, 2], [1, 0]]
```

The handling of indices here and in future questions requires special attention! Matrices are represented in Python using lists of lists which means that indices range from 0 to $n-1$. However, the construction of $\hat{A}$ considers indices to range from 1 to n. See the example above.

Complete the function `compute_witness_unique(A,B)` that computes a witness matrix under the assumption that each $C[i][j]$ has a unique witness.

```
In [10]: compute_witness_unique(A,B)
Out[10]: [[2, 1], [0, 1]]
```

What is the complexity of this algorithm? (Write the answer as a comment in the code.)

Since there is no guarantee that each $C[i][j]$ has a unique witness, we use randomization to achieve the effect of such a guarantee for a sufficiently large number of entries in C. Suppose that an entry $C[i][j]$ has a number $\alpha \geq 2$ of witnesses. Let r be an integer such that $n/2 \leq \alpha \cdot r \leq n$. We claim that a random set of indices $R \subseteq \{1, \ldots, n\}$ of cardinality r is very likely to contain a unique witness for $C[i][j]$. Indeed, the following is true :

$$\text{The probability that } R \text{ contains } \textit{exactly one} \text{ witness is greater than } \frac{1}{2e} \tag{1}$$

Assuming that the set R contains a unique witness for $C[i][j]$, we extend the technique in Question 6 to compute this witness. Let A^R be the matrix defined by : for all $i, k \in \{1, \ldots, n\}$

$$A^R[i][k] = \begin{cases} A[i][k], & \text{if } k \in R, \\ 0, & \text{otherwise} \end{cases}$$

The difference between the matrix A^R and A is that each *column* of A^R corresponding to an index that is not in R is turned into an all-zero vector (the other entries are the same as in A). For instance,

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \quad A^{\{1,2\}} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Also, let B^R be the matrix defined by : for all $k, j \in \{1, \ldots, n\}$

$$B^R[k][j] = \begin{cases} B[k][j], & \text{if } k \in R, \\ 0, & \text{otherwise} \end{cases}$$

Each *row* of B^R corresponding to an index that is not in R is turned into an all-zero vector, e.g.,

$$B = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \quad B^{\{1,2\}} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Then, if the entry $C[i][j]$ has a unique witness in R and $W^R = \widehat{A^R} \cdot B^R$, then $W^R[i][j]$ is the index of this unique witness. The matrix $\widehat{A^R}$ is defined starting from A^R as $\hat{A}$ is defined starting from A.

Question 7. [2 points] Complete the function `nullify_columns(A,R)` that returns a matrix where each column of A corresponding to an index that is not in R is turned into an all-zero vector. Complete the function `nullify_rows(A,R)` that turns rows not in R to all-zero vectors.

```
In [11]: A = [[1,1,1],[1,1,0],[0,1,1]]
In [12]: B = [[1,0,1],[0,1,0],[0,1,1]]
In [13]: nullify_columns(A,[1,2])
Out[13]: [[1, 1, 0], [1, 1, 0], [0, 1, 0]]
In [14]: nullify_rows(B,[1,2])
Out[14]: [[1, 0, 1], [0, 1, 0], [0, 0, 0]]
```

Complete the function `compute_witness_restricted(A,B,R)` that returns the matrix W^R defined above, which is a witness matrix assuming that each entry $C[i][j]$ has a unique witness in R (do not forget to call the `expose_column` function for the $\widehat{}$ operator).

```
In [15]: compute_witness_restricted(A,B,[1,2])
Out[15]: [[1, 2, 1], [1, 2, 1], [0, 2, 0]]
```

What is the complexity of this algorithm? (Write the answer as a comment in the code.)

As stated in (1), there is a constant probability that a randomly chosen set R of size r has a unique witness for an entry $C[i][j]$ (assuming that $n/2 \le \alpha \cdot r \le n$). Therefore, repeating the random choice of the set R for $(2 \cdot e \cdot \log n)$ many times makes it extremely unlikely that the set would not contain exactly one witness.

Question 8. [1 point] How unlikely? Give an upper bound for the probability of failure. (Write the answer as a comment in the code.)

Question 9. [1 point] Complete the function `sample(r,n)` that samples a subset of size r of $\{1,\dots,n\}$, uniformly at random. You can start with an array X of length n with $X[i] = i+1$ for each i (the $i+1$ is because we sample from $\{1,\dots,n\}$ and not $\{0,\dots,n-1\}$), shuffle the array, and return its first r elements. To shuffle an array, you can traverse its indices from left to right, and for each i, generate a random value $j \in \{i,\dots n-1\}$, and swap $X[i]$ with $X[j]$. This is the Fisher–Yates Shuffle Algorithm.

```
In [16]: sample(3,10)
Out[16]: [8, 1, 7]
```

We have all the ingredients to define the randomized algorithm for solving the BPWM problem :

```
∘ $W = -A \cdot B$
∘ for each $t \in \{0,\dots,\lfloor log\ n \rfloor\}$ do
    ∘ $r = 2^t$
    ∘ repeat $2 \cdot e \cdot \log n$ times
        ∘ sample a set $R \subseteq \{1,\dots,n\}$ of size $r$ uniformly at random (Question 9)
        ∘ compute $W^R = \widehat{A^R} \cdot B^R$ (Question 7)
        ∘ for all $i,j$, if $W[i][j] < 0$ and $W^R[i][j]$ is a valid witness for $C[i][j]$ (use the definition),
          then $W[i][j] = W^R[i][j]$
∘ for all $i,j$, if $W[i][j] < 0$, then find the witness for $C[i][j]$ using the straightforward approach
  (enumerate all indices $k$).
```

The initial setting of W ensures that the only negative entries are those where the value of $C[i][j]$ is non-zero and there is a need to find a witness. The brute-force search in the last step for the witnesses not identified by the randomized strategy ensures that the algorithm is Las Vegas.

Question 10 (2 points). Complete the function `compute_witness_random(A,B)` that returns the matrix W computed using the algorithm described above.

```
In [17]: compute_witness_random(A,B)
Out[17]: [[1, 3, 1], [1, 2, 1], [0, 3, 3]]
```

What is the expected running time of this algorithm in function of the complexity of matrix multiplication? (Write the answer as a comment in the code.)