1. Let P be a set of n points $p_1, p_2, \ldots, p_n$ in the 2-dimensional plane with positive integer coordinates. That is, $p_i = (x_i, y_i)$ where $x_i, y_i > 0$ and x_i, y_i are integers. We will assume for simplicity that no two points have the same coordinates. Each point p_i defines a rectangle R_i with one corner being p_i and the opposite corner being the origin $(0, 0)$. A point p_i dominates a point p_j if p_j is inside the rectangle R_i; in other words $x_j \leq x_j$ and $y_j \leq y_i$. A point $p_i \in P$ is *undominated* if no point in P dominates it. Let $Q \subseteq P$ be the undominated points in P. See the figure.

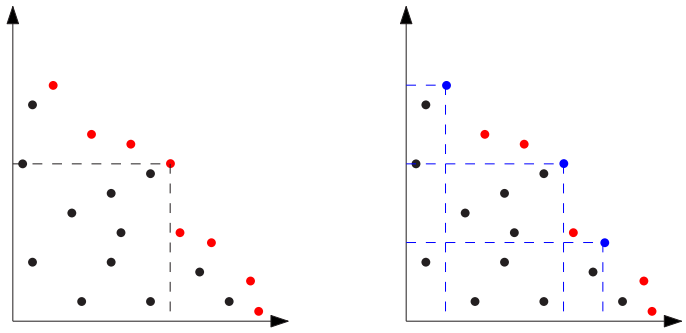


Figure 1. (a) A set of points with positive coordinates with the undominated points shown in red. (b) 3 points from Q which dominate a total of 14 points (including themselves) .

(3 pts) Given P describe an efficient algorithm that outputs Q the set of all undominated points in P. A trivial algorithm runs in $O(n^2)$ time and there is a relatively simple $O(n \log n)$ time algorithm for this.

Solution (standard): Notice that the undominated points are the outer corners of a "staircase" walking above P. We can formalize this intuition with the following characterization: A point $(p_x, p_y) \in P$ is undominated if and only if every other point $(q_x, q_y) \in P$ with $q_y \geq p_y$ has $q_x < p_x$. This leads to the following $O(n \log n)$ algorithm that walks down the staircase:

```
UNDOMINATED(P[1..n]):
  P' ← P sorted ascending by y-coordinate;
          tiebreak with x ⟨⟨In O(n log n) time⟩⟩
  x ← 0          ⟨⟨Largest x-coordinate seen so far⟩⟩
  Q ← [ ]
  for i ← n down to 1
      (p_x, p_y) ← P'[i]
      if p_x > x
          x ← p_x
          Q.add((p_x, p_y))
  return Q
```

We can also do better![a] This problem is closely related to the problem of computing a structure called the *convex hull* of a set of points, and we can mimic the approaches of two seminal papers to solve this problem. The first is based on the "Quickhull" algorithm by Timothy Chan, Jack Snoeyink, and Chee-Keng Yap. The second is based on Chan's "Shatter Hull" grouping algorithm. These solutions are presented at the

end of this document. ■

[a] You don't need to understand either of the fast solutions; we are presenting these because they might be of interest.

Rubric: 3pts for runtime of $O(n \log n)$ or better. -1pt for minor mistake.

(7 pts) Given P and a subset $P' \subseteq P$ we let Dominated(P') be the set of all points in P that are dominated by P'. Given P and an integer k we would like to find a subset $P' \subseteq P$ such that $|P'| \leq k$ and P' maximizes the number of points it dominates.

- (1 pt) Argue that for any k there is always a subset of Q that is an optimum solution.

Solution: We'll write $p \preccurlyeq q$ to mean q dominates p. Note that the relation is transitive: $p \preccurlyeq q$ and $q \preccurlyeq r$ implies $p \preccurlyeq r$. Consider some $P' \subseteq P$. Every point in P is dominated by some point in Q (proof below), so for every $p \in P'$, replace it by a $q_p \in Q$ that dominates it to construct a set $P'' \subseteq Q$. Note that $|P''| \leq |P'|$. Then, for every $p \in P'$, if p dominates r, then $r \preccurlyeq p \preccurlyeq q_p \in P''$. So, Dominated$(P')$ $\subseteq$ Dominated(P''). Therefore, if P' dominates as many elements as possible with $|P'| \leq k$, so does P''.

Claim 1. *Every $p \in P$ is dominated by some $q \in Q$.*[a]

Proof: Intuitively, this is true because starting from any point in P, we can "follow it up" until we arrive at a point in Q.

To be formal, we'll induct on $n = |P|$. If $n = 1$, then $P = Q$ and the single element is dominated by itself.

Now suppose all sets P' with $|P'| < n$ have every element dominated by some point in Q'. Let $p \in P$. If $p \in Q$, we're done: $p \preccurlyeq p \in Q$. Otherwise, consider the set $P' = \{q \in P : q \neq p, p \preccurlyeq q\}$. This is a strict subset, and it's nonempty (otherwise p would be in Q): $1 \leq |P'| < n$. Let $p' \in P'$. By the inductive hypothesis, there is some $q' \in Q'$ such that $q' \succcurlyeq p'$, and since $p' \succcurlyeq p, q' \succcurlyeq p$.

Now we argue that $q' \in Q$. Since q' dominates p, any point that dominates it must dominate p (i.e. it must be in P'). But no point in P' dominates q' since $q' \in Q'$. Hence, $q' \in Q$. □

■

[a] A formal proof of this fact is not necessary for credit.

Rubric: 1pt for a correct substitution argument.

- (6 pts) Using the preceding fact and an ordering of Q according to the x-coordinate, describe an efficient dynamic programming based algorithm to find the maximum number of points that a k-element subset of Q can dominate. Technically you do not need the preceding part to solve this part, but it helps you to understand the problem better. You may find the figures useful. Any solution with a running time of $O(n^3)$ or better will get full credit. Any polynomial-time algorithm will get at least 80%.

Solution (3 parameters): Obtain Q as above, and sort it ascending by x-coordinate (so descending by y-coordinate). Let $h = |Q|$. Add a sentinel value $Q[0] = (0, 0)$.

Define *MaxDom*(i, j, ℓ) as the maximum number of points that can be dominated by up to ℓ points from $Q[j..h]$ that are not dominated by $Q[i]$. The function obeys the recurrence:

$$\textit{MaxDom}(i,j,\ell) = \begin{cases} 0 & \text{if } j > h \text{ or } \ell = 0 \\ \max \left\{ \begin{array}{l} \textit{MaxDom}(i, j+1, \ell), \\ \textit{MaxDom}(j, j+1, \ell-1) \\ \quad + \textit{DomPast}(i,j) \end{array} \right\} & \text{otherwise} \end{cases}$$

Where *DomPast*(i, j) is the number of points dominated by $Q[j]$ but not by $Q[i]$. The size of the largest dominated set is given by *MaxDom*$(0, 1, k)$.

We precompute *DomPast* in the standard way: for every pair of distinct indices (i, j), we iterate through all points in P and count how many are dominated by $Q[j]$ but not $Q[i]$. This takes $O(nh^2)$ time. We use a 3-dimensional array *MaxDom*$[0 .. h+1, 1 .. h+1, 0 .. k]$ to memoize our original function.

```
MaxDom(P[1..n], k):
  Q ← UNDOMINATED(P) ⟪O(n log h)⟫
  Sort Q by ascending x-coordinate
  Q[0] ← (0, 0)
  MaxDom ← 0 in all entries ⟪Covers base cases⟫
  Precompute DomPast[0..h, 0..h] ⟪O(nh²)⟫
  for i ← h down to 0
      for j ← h down to 1
          for ℓ ← 1 to k
              skip ← MaxDom[i, j+1, ℓ]
              take ← MaxDom[j, j+1, ℓ−1] + DomPast[i, j]
              MaxDom[i, j, ℓ] ← max{skip, take}
  return MaxDom[0, 1, k]
```

Note that if $k \geq h$, we can immediately return n (we choose all undominated points). The time to fill in *MaxDom* is $O(h^2 k)$, so the running time of *MaxDom*$(P[1 .. n], k)$ is dominated by the precomputation step: $O(nh^2)$.

(It is possible to precompute *DomPast* much more efficiently – see the end of these solutions for how. This makes computing *MaxDom* dominate the runtime, for worst-case $O(n^3)$, though this too can be optimized to $O(nk \log k)$ using some advanced machinery. None of these optimizations are required for full credit.) ■

Solution (2 parameters): Compute Q and *DomPast* as above.

Let *MaxDom*(i, j) be the maximum number of points that can be dominated by up to j points from $Q[1 .. i]$. We want *MaxDom*(h, k). The function satisfies

the following recurrence:

$$\mathit{MaxDom}(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max \left\{ \begin{array}{l} \mathit{MaxDom}(i-1,j), \\ \max\limits_{1 \le l \le i-1} \left\{ \begin{array}{l} \mathit{MaxDom}(l, j-1) \\ + \mathit{DomPast}(l,i) \end{array} \right\} \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a 2D array $\mathit{MaxDom}[0..h, 0..k]$, and evaluate it in order of increasing i in the outer loop and in arbitrary order in the inner loop. There are $O(hk)$ subproblems, each requiring $O(h)$ time to compute. The resulting algorithm runs in $O(h^2k)$ time plus time required to precompute *DomPast*, for $O(nh^2)$ overall.

The following recurrence is incorrect:

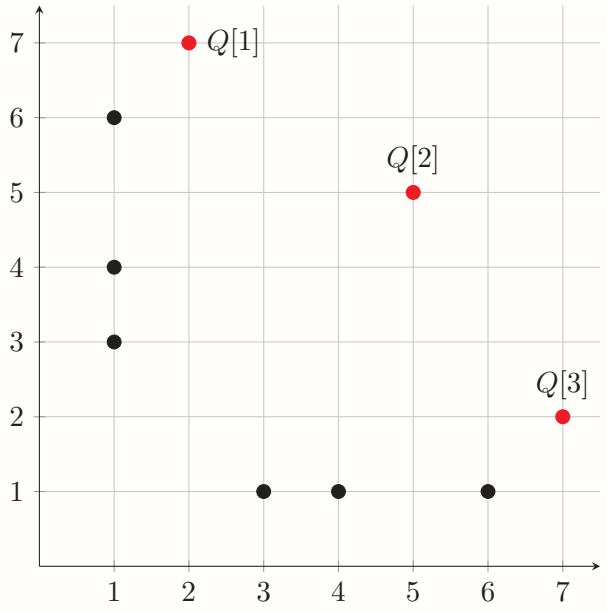
$$\mathit{MaxDom}(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max \left\{ \begin{array}{l} \mathit{MaxDom}(i-1,j), \\ \mathit{MaxDom}(i-1,j-1) \\ \quad + |\mathit{Dom}(i)| \end{array} \right\} & \text{otherwise} \end{cases}$$

where $\mathit{Dom}(i)$ is the set of points dominated by $Q[i]$, which can be precomputed. This might appear to lead to an $O(n^2)$ solution, but there are two issues. First, $\mathit{Dom}(i)$ may overlap with other *Dom* sets, because a point can be dominated by multiple undominated points. Thus the above recurrence overcounts dominated points. **But there is a larger issue:**

Suppose we try to avoid overcounting by modifying *MaxDom* to store sets and unioning sets, preventing duplicate points. Specifically, let $\mathit{MaxDomSet}(i,j)$ be the *largest set* of points dominated by up to j points from $Q[1..i]$. We want $|\mathit{MaxDomSet}(h,k)|$. The function "satisfies" the following recurrence, where the "max" of two sets here returns the larger set (arbitrarily chosen if tied [a]):

$$\mathit{MaxDomSet}(i,j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ \max \left\{ \begin{array}{l} \mathit{MaxDomSet}(i-1,j), \\ \mathit{MaxDomSet}(i-1,j-1) \\ \quad \cup \mathit{Dom}(i) \end{array} \right\} & \text{otherwise} \end{cases}$$

This no longer overcounts points. However, this is still incorrect! Consider the example given by the points below with $k = 2$:



For convenience, assume $\mathit{MaxDomSet}(i, j)$ does not include points from Q [b]. According to the English definition of *MaxDomSet*,

$$\mathit{MaxDomSet}(3, 2) = \{(1, 3), (1, 4), (1, 6), (3, 1), (4, 1), (6, 1)\},$$

as $Q[1]$ and $Q[3]$ together dominate all the black points. Moreover,

$$\mathit{MaxDomSet}(2, 1) = \mathit{Dom}(2) = \{(1, 3), (1, 4), (3, 1), (4, 1)\},$$

since $Q[2]$ dominates more points than any other single point, so

$$\begin{aligned}\mathit{MaxDomSet}(2, 1) \cup \mathit{Dom}(3) &= \mathit{Dom}(2) \cup \mathit{Dom}(3) \\ &= \{(1, 3), (1, 4), (3, 1), (4, 1), (6, 1)\},\end{aligned}$$

and also

$$\mathit{MaxDomSet}(2, 2) = \mathit{Dom}(1) \cup \mathit{Dom}(2) = \{(1, 3), (1, 4), (1, 6), (3, 1), (4, 1)\}.$$

But according to the recurrence,

$$\mathit{MaxDomSet}(3, 2) = \max \left\{ \begin{matrix} \mathit{MaxDomSet}(2, 2), \\ \mathit{MaxDomSet}(2, 1) \cup \mathit{Dom}(3) \end{matrix} \right\}$$

This is a maximum between two sets of size 5, but $\mathit{MaxDomSet}(3, 2)$ should be a set of size 6. Therefore, the recurrence gives the wrong answer.

Indeed, this recurrence is fatally flawed because it implicitly assumes that the optimal subset of $Q[1 .. i]$ containing the point $Q[i]$ is the optimal subset of $Q[1 .. i - 1]$ with $Q[i]$ added. As we saw above, this is not true! $Q[i]$ may overlap heavily (in terms of points dominated) with the optimal subset of $Q[1 .. i - 1]$, while overlapping much less with some suboptimal subset of $Q[1 .. i - 1]$, to the

point where the suboptimal subset of $Q[1..i-1]$ becomes better when combined with $Q[i]$.

Hence, we must try all optimal subsets of $Q[1..l]$ over $1 \leq i \leq i-1$ to combine with $Q[i]$, as seen in the correct 2 parameter recurrence. (The 3 parameter recurrence does the same but uses the extra parameter to try all these possibilities rather than doing it inside the recurrence.) ■

[a] This doesn't actually work. Sets require great care – ideally just don't use them in dynamic programming algorithms. Using sets also increases the runtime by a factor of n. Instead of using sets, one should store only the used point in Q with the largest x-coordinate, in addition to the size of the set, and use *DomPast*. But here this still doesn't work, for the same reasons we'll see.

[b] If we need them, they are accounted for by the value of j – we can add j to the resulting size.

Rubric: Standard DP rubric, scaled to 6 pts for an $O(n^3)$ (or faster) algorithm and 5 pts for a slower polynomial time algorithm.

2. A sequence $A = a_1, a_2, \ldots, a_n$ is *palindromic* if A and its reverse are the same sequence, that is, $a_{n-i+1} = a_i$ for $i = 1, 2, \ldots, n$. Examples of palindromic strings (recall strings are sequences over an alphabet Σ) are A, MALAYALAM, KAYAK, 374473, 47374. In the lab we saw the following problem: given a sequence A, find the length of a longest palindromic subsequence of A. You may want to review that before working on this problem. As an additional exercise, you may want to figure out how to compute the optimum value in $O(n)$ space (this is not to submit).

   - (5 pts) Call a palindrome w interesting if no two adjacent characters in w are the same. For example 47374 and KAYAK are interesting while 44344 and 374473 are *not* interesting. Describe an efficient algorithm that outputs the length of a longest interesting palindromic subsequence of a given sequence A and an alphabet Σ (which is a set of symbols A may choose from). Express the running time in terms of your input.

Solution: Define $\Sigma' = \Sigma \cup \{\square\}$, where $\square$ is a character not in Σ. Let $\mathit{LIPS}(i, j, c)$ denote the length of a longest interesting palindromic subsequence of $A[i \,..\, j]$ that does *not* start (or end) with c. The function satisfies the following recurrence:

$$\mathit{LIPS}(i,j,c) = \begin{cases} 0 & \text{if } i > j \\ [A[i] \neq c] & \text{if } i = j \\ 2 + \mathit{LIPS}(i+1, j-1, A[i])^a & \text{if } A[i] = A[j] \neq c \text{ and} \\ & \mathit{LIPS}(i+1, j-1, A[i]) > 0 \\ \max \begin{Bmatrix} \mathit{LIPS}(i+1, j, c) \\ \mathit{LIPS}(i, j-1, c) \end{Bmatrix} & \text{otherwise} \end{cases}$$

Where the notation $[\mathit{cond}]$ evaluates to 1 if *cond* is true; 0 otherwise. Note the condition on the third case: we can only add both characters if the subsequence between them could be nonempty. The solution to the longest interesting palindromic subsequence is given by $\mathit{LIPS}(1, n, \square)$.

We memoize our function using a 3-dimensional array $\mathit{LIPS}[1 \,..\, n+1, 1 \,..\, n, \Sigma']$, indexed by two positions and a character. $\mathit{LIPS}[i, j, c]$ depends on calls with a higher first parameter and/or a lower second parameter, so we can evaluate the first index downwards in the outermost loop, the second index upwards in the middle loop, and the third index in any order in the innermost loop.

```
LIPS(A[1..n]):
  LIPS ← 0 in all entries
  for i ← 1 to n                          ⟨⟨Base cases⟩⟩
      for c ∈ Σ'
          LIPS[i, i, c] ← [A[i] ≠ c]
  for i ← n down to 1
      for j ← i + 1 to n
          for c ∈ Σ'
              innerLen ← LIPS(i + 1, j − 1, A[i])
              if A[i] = A[j], A[i] ≠ c, and innerLen > 0
                  LIPS[i, j, c] ← 2 + innerLen
              else
                  LIPS[i, j, c] ← max {LIPS[i + 1, j, c], LIPS[i, j − 1, c]}
  return LIPS[1, n, □]
```

Letting $k = |\Sigma|$, $LIPS(A[1..n])$ runs in $O(n^2k)$ time. ■

[a] See Lab 7 for a proof of why we can greedily add both $A[i]$ and $A[j]$ to our subsequence.

Rubric: Standard DP rubric, scaled to 5 pts for an $O(n^2k)$ algorithm and 3 pts for a slower polynomial time algorithm.

- (5 pts) Let $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_n$ be two sequences. Describe an efficient algorithm to compute the length of the longest *common palindromic subsequence* of X and Y. That is, if the answer is h then there is a palindromic sequence Z of length h such that Z is a subsequence of X *and* Z is a subsequence of Y. For example if X is the string afbcdfca and Y is the string bcadfcgyfka then your output should be 5 corresponding to the palindrome afcfa that is a common subsequence of both X and Y.

Solution: Let $LCPS(i, j, k, \ell)$ denote the length of the longest common palindrome subsequence of $X[i..j]$ and $Y[k..\ell]$. The function then satisfies the following recurrence:

$$LCPS(i,j,k,\ell) = \begin{cases} 0 & \text{if } i > j \text{ or } k > \ell \\ [X[i] = Y[k]] & \text{if } i = j \text{ and } k = \ell \\ \max \left\{ \begin{array}{l} LCPS(i+1, j, k, \ell) \\ LCPS(i, j-1, k, \ell) \\ LCPS(i, j, k+1, \ell) \\ LCPS(i, j, k, \ell-1) \\ \left(\begin{array}{l} 2 + LCPS(i+1, j-1, k+1, \ell-1) \\ \text{if } X[i] = X[j] = Y[k] = Y[\ell] \\ \quad \text{and } i < j \text{ and } k < l \end{array} \right) \end{array} \right\} & \text{otherwise} \end{cases}$$

Our final answer is given by $LCPS(1, m, 1, n)$.

We can memoize this with a 4-d array $LCPS[1..m+1, 0..m, 1..n+1, 0..n]$. Subproblems depend on those with larger, smaller, larger, and smaller first,

second, third, and fourth parameters, respectively, so we can evaluate them descending, ascending, descending, and ascending, respectively.

```
LCPS(X[1..m], Y[1..n]):
  LCPS ← 0 in all entries
  for i ← 1 to m                ⟨⟨Base Cases⟩⟩
      for k ← 1 to n
          LCPS[i, i, k, k] ← [X[i] = Y[k]]

  for i ← m − 1 down to 1       ⟨⟨i = m ⇒ j ≤ m = i; handled above⟩⟩
      for j ← i + 1 to m
          for k ← n − 1 down to 1
              for ℓ ← k + 1 to n
                  if X[i] = X[j] = Y[k] = Y[ℓ]
                      LCPS[i, j, k, ℓ] ← 2 + LCPS[i + 1, j − 1, k + 1, ℓ − 1]
                  skip ← max { LCPS[i + 1, j, k, ℓ]
                             { LCPS[i, j − 1, k, ℓ]
                             { LCPS[i, j, k + 1, ℓ]
                             { LCPS[i, j, k, ℓ − 1]
                  LCPS[i, j, k, ℓ] ← max {LCPS[i, j, k, ℓ], skip}
  return LCPS[1, m, 1, n]
```

As we do constant work each iteration, $LCPS(X[1..m], Y[1..n])$ runs in $O(m^2n^2)$ time. ■

Rubric: Standard DP rubric, scaled to 5 pts for an $O(m^2n^2)$ algorithm and 3 pts for a slower polynomial time algorithm.

Standard dynamic programming rubric. 10 points =

- 3 points for a clear and correct English description of the recursive function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.)
  - − 1 for naming the function "OPT" or "DP" or any single letter.
  - – No credit if the description is inconsistent with the recurrence.
  - – No credit if the description does not explicitly describe how the function value depends on the named input parameters.
  - – No credit if the description refers to internal states of the eventual dynamic programming algorithm, like "the current index" or "the best score so far". The function must have a well-defined value that depends *only* on its input parameters (and constant global variables).
  - – An English explanation of the *recurrence* or *algorithm* does not qualify. We want a description of *what* your function returns; points for an explanation of *how* that value is computed are assigned in other items.
- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.

+ 3 for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error.
− 2 for greedy optimizations without proof, even if they are correct.
– **No credit for iterative details if the recursive case(s) are incorrect.**

- 3 points for iterative details
  + 1 for describing an appropriate memoization data structure. **Hash tables are NOT an appropriate memoization data structure!**
  + 1 for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested for loops, be sure to specify the nesting order. (In particular, if you draw a rectangle for a 2d array, be sure to label and direct the row and column indices.)
  + 1 for correct time analysis. (It is not necessary to state a space bound.)

- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem specifically says otherwise.
- Iterative pseudocode is ***not*** required for full credit, provided the other details of your solution are clear and correct. We usually give two official solutions, one with pseudocode and one without.

  If your solution does includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. But you ***do*** still need and English description of the underlying recursive function (or equivalently, the contents of the memoization structure). ***Perfectly correct iterative pseudocode, with no explanation or time analysis, is worth at most 6 points out of 10***, or at most 5 points out of 10 if you name the memoization array "DP".
- Partial credit for incomplete solutions depends on the running time of the ***best possible*** completion (up to the target running time). For example, consider a solution that contains *only* a clear English description of a function, with no recurrence or iterative details. If the described function *can* be developed into an algorithm with the target running time, the solution is worth 3 points; however, if the function leads to an algorithm that is slower than the target time by a factor of n, the solution could be worth only 2 points ($= 70\%$ of 3, rounded).

Optimizations

Once again, you don't need to understand these, but they might be interesting.

Problem 1a

Solution (Quickhull): Let $n = |P|$ and $h = |Q|$. We'll use a divide and conquer algorithm to improve the running time to $O(n \log h)$. Naively, one would split the points into equal size sets P_- and P_+, divided across some line $x = x_{\text{mid}}$, and recursively compute $\textsc{Undominated}(P_-)$ and $\textsc{Undominated}(P_+)$. Points in P_+ can't be dominated by those in P_-, so that call is correct, but points in P_- can be dominated by points in P_+. We see that a point in P_- is undominated in P iff it's undominated in P_- and it has a larger y-coordinate than every point in P_+. So to find the set of undominated points, we'll...

1. Check if $n \leq 1$. If so, return P.

2. Compute p_m, the point with median x-coordinate.
3. Split P into P_+ and P_-, the points with greater and or equal to and smaller x-coodrinates that p_m, respectively.
4. Let $y_{+,\max}$ be the largest y-coordinate in P_+. Remove all points in P_- with y-coordinate less than or equal to $y_{+,\max}$ (these are dominated by some point in P_+).
5. Recursively compute the set of undominated points in P_- and P_+.
6. Return the concatenation of both lists.

We can find p_m, P_- and P_+ with linear time selection and partitioning. It takes $O(n)$ time find the largest y-coordinate in P_+ and to prune P_-.

If $h = 0$, $n = 0$, so our running time is constant. If $h = 1$, P_- becomes empty after pruning, so we will have exactly one nontrivial recursive call and our running time becomes $n + n/2 + n/4 \cdots = O(n)$. Our recurrence is then

$$\begin{aligned} T(n, 0) &= 1 \\ T(n, 1) &= O(n) \\ T(n, h) &\le T(n/2, h_1) + T(n/2, h - h_1) + O(n) \end{aligned}$$

Where h_1 is the number of undominated points in P_-. So $T(n, h) = O(n \log h)$.

Proof sketch: Base cases clearly work. Suppose $T(n', h') \le Mn' \log(h')$ for $1 \le h' \le n' < n, M \ge \frac{1}{2}$. Then, $T(n, h)$

$$\begin{aligned} &\le \max_{2 \le h_1 \le h-2} (T(n/2, h_1) + T(n/2, h - h_1)) + n && (h_1 \le 1 \text{ won't maximize}) \\ &\le \max_{h_1 \in (0,h)} \left(\frac{Mn}{2} \log(h_1) + \frac{Mn}{2} \log(h - h_1) \right) + n \\ &= \frac{Mn}{2} \max_{h_1 \in (0,h)} \log(h_1(h - h_1)) + n \\ &= \frac{Mn}{2} \log(h^2/4) + n && (\text{Maximum at } h_1 = h/2) \\ &= Mn(\log h - 2) + n \le Mn \log(h) \end{aligned}$$

So $T(n, h)$ is $O(n \log h)$. ■

Solution (Shatter Hull): We begin by defining a subroutine CANDIDATEPT that takes as input a set of points $Q_j[1 \,..\, k]$ sorted ascending by x-coordinate and descending by y-coordinate and a query point q. We return the rightmost point $p \in Q_j$ such that $p.x < q.x$ and $p.y > q.y$ (or $(-\infty, -\infty)$ if no such point exists). Since the points are sorted, this can be done with binary search in $O(\log k)$ time:

```
CandidatePt(Q_j[1..k], q):
  if k < 374: Brute Force, return (−∞, −∞) if no feasible point
  m ← Q_j[⌈k/2⌉]
  if m_x < q_x and m_y > q_y
      return CandidatePt(Q_j[⌈k/2⌉..k], q)
  else
      return CandidatePt(Q_j[1..⌈k/2⌉], q)
```

The algorithm below assumes we know h in advance. For now, just assume $|Q| = h$ is whispered to us by a chorus of harmonizing raccoons channeling the spirit of your favorite Taylor Swift album, and we'll worry about finding h later.

On a high level, the algorithm works by partitioning the points and computing several "mini-staircases" for each subset. We can then build the final set of undominated points from right to left, noticing that the next undominated point after a current point q is the rightmost point p such that $p.x < q.x$ and $p.y > q.y$ (the same problem we solved in CandidatePt!).

```
Undominated(P[1..n], h):
  Partition P into ⌈n/h⌉ subsets of size ≈ h: P_1, ..., P_⌈n/h⌉
  for j ← 1 to ⌈n/h⌉
      Q_j ← Undominated points of P_j
      ⟨⟨Using the O(n log n) algorithm, sorted ascending by x coordinate⟩⟩
  q ← rightmost point
  Q ← [ ]
  for i ← 1 to h
      Q.add(q)
      ⟨⟨The leftmost undominated point will always be the topmost point⟩⟩
      if q = topmost point: return Q
      next ← topmost point
      for j ← 1 to ⌈n/h⌉
          p ← CandidatePt(Q_j, q)
          if p.x > next.x or p dominates next: next ← p
      q ← next
  return Fail ⟨⟨We'll explain why later⟩⟩
```

Runtime: Computing each Q_j takes $O(h \log h)$ time, since $|P_k| \leq h$ by construction. As we do this $O(\frac{n}{h})$ times, computing all Q_js takes $O(\frac{n}{h} \cdot h \log h) = O(n \log h)$ time.

The total runtime of the main loop is

$$O(\overbrace{h}^{\text{outer iterations}} \cdot \overbrace{n/h}^{\text{inner iterations}} \cdot \overbrace{\log h}^{\text{CandidatePt}}) = O(n \log h)$$

Thus the final runtime is given by $O(n \log h) + O(n \log h) = O(n \log h)$.

I forgot that h existed: Lamentably, raccoons don't know Taylor Swift, so we do need to figure out the value of h ourselves. Notice that the algorithm fails if the current point q never reaches the topmost point–i.e. if we underestimate h. So, we can try a sequence of values until we do not Fail: We try

$$\hat{h} = 2^{2^1}, 2^{2^2}, 2^{2^3}, 2^{2^4}, \ldots, 2^{2^i}, \ldots$$

until we we do not Fail. This means we will keep iterating until we try $2^{2^{\lceil \lg \lg h \rceil}}$. Notice that

this implies a final runtime of

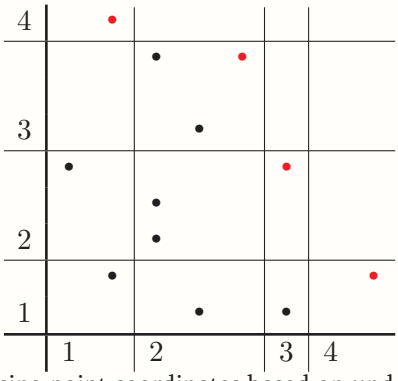
$$O\left(\sum_{i=1}^{\lceil \lg \lg h \rceil} n \lg\left(2^{2^i}\right)\right) = O\left(\sum_{i=1}^{\lceil \lg \lg h \rceil} n 2^i\right) = O\left(n 2^{\lg \lg h}\right) = O(n \log h)$$

■

Problem 1b

Solution (fast): Assume we have $Q[1..h]$ computed, ordered by ascending x-coordinate. We describe an optimization for precomputing *DomPast*.

Let $h = |Q|$. Let X and Y be sets of the x and y-coordinates of Q. Relabel the x and y-coordinates of the points in P as follows: Replace $p = (p_x, p_y)$ with $p' = (i, j)$ where i is the rank of the largest $x \in X$ with $p_x \leq x$, and j is the rank of the largest $y \in Y$ with $p_y \leq y$. This can be done in $O(n \log h)$ time by performing a binary search on X and Y for each $p \in P$. Note that this does not affect our final answer; each undominated point dominates exactly the same points as before.



$$P = \begin{bmatrix} (1,5),(2,2),(2,9),(3,3), \\ (3,4),(3,8),(4,1),(4,6), \\ (5,8),(6,1),(6,5),(8,2), \end{bmatrix}$$

$$\Downarrow$$

$$P = \begin{bmatrix} (1,2),(1,1),(1,4),(2,2), \\ (2,2),(2,3),(2,1),(2,3), \\ (2,3),(3,1),(3,2),(4,1), \end{bmatrix}$$

Compressing point coordinates based on undominated points. Numbers beside the axes are the new coordinates.

Now $P \subseteq \{1, ..., h\} \times \{1, ..., h\}$.

Define $\mathit{Dom}(x, y)$ to be the number of points in P dominated by (x, y). Note that (x, y) does not have to be a point in P. *Dom* obeys the recurrence relation

$$\mathit{Dom}(x,y) = \begin{cases} 0 & \text{if } x = 0 \text{ or } y = 0 \\ \mathit{Dom}(x-1,y) + \mathit{Dom}(x,y-1) & \\ \quad -\mathit{Dom}(x-1,y-1) + \#(x,y) \in P & \text{otherwise} \end{cases}$$

Where $\#(x, y) \in P$ is the number of points with coordinates (x, y) in the list (we treat P as a list now since it can have duplicates after coordinate compression). We can memoize this into an array $\mathit{Dom}[0..h, 0..h]$. Calls depend on smaller x and/or y-coordinates, so we can evaluate the first and second index in ascending order in $O(h^2)$ time:

```
ComputeDomMatrix(P):
  ⟨⟨For constant time checking of #(x, y) ∈ P⟩⟩
  Grid[1 .. h, 1 .. h] ← 0 in all entries
  Dom[0 .. h, 0 .. h] ← 0 in all entries ⟨⟨Covers base cases⟩⟩
  for (x, y) in P
      Grid[x, y] ← Grid[x, y] + 1

  for x ← 1 to h
      for y ← 1 to h
          Dom[x, y] ← Dom[x − 1, y] + Dom[x, y − 1]
                      −Dom[x − 1, y − 1] + Grid[x, y]
  return Dom
```

Let $a = Q[i], b = Q[j]$. To compute *DomPast*(i, j), we return the number of points dominated by b minus the number of points dominated by a and b: *Dom*$[b.x, b.y] -$ *Dom*$[b.x, a.y]$. Our total runtime then improves to $O(h^2k + n \log h)$. ■