it is often necessary to design algorithms that are simple, and easily modified if problem parameters and specifications are slightly modified. Fortunately, most of the algorithms that we will discuss in this class are quite simple, and are easy to modify subject to small problem variations.

1.7 Model of Computation

Another goal that we will have in this course is that our analysis be as independent as possible of the variations in machine, operating system, compiler, or programming language. Unlike programs, algorithms to be understood primarily by people (i.e. programmers) and not machines. Thus gives us quite a bit of flexibility in how we present our algorithms, and many low-level details may be omitted (since it will be the job of the programmer who implements the algorithm to fill them in).

But, in order to say anything meaningful about our algorithms, it will be important for us to settle on a mathematical model of computation. Ideally this model should be a reasonable abstraction of a standard generic single-processor machine. We call this model a *random access machine* or RAM.

A RAM is an idealized machine with an infinitely large random-access memory. Instructions are executed one-by-one (there is no parallelism). Each instruction involves performing some basic operation on two values in the machines memory (which might be characters or integers; let's avoid floating point for now). Basic operations include things like assigning a value to a variable, computing any basic arithmetic operation (+, - , $\times$, integer division) on integer values of any size, performing any comparison (e.g. $x \leq 5$) or boolean operations, accessing an element of an array (e.g. $A[10]$). We assume that each basic operation takes the same constant time to execute.

This model seems to go a good job of describing the computational power of most modern (nonparallel) machines. It does not model some elements, such as efficiency due to locality of reference, as described in the previous lecture. There are some "loop-holes" (or hidden ways of subverting the rules) to beware of. For example, the model would allow you to add two numbers that contain a billion digits in constant time. Thus, it is theoretically possible to derive nonsensical results in the form of efficient RAM programs that cannot be implemented efficiently on any machine. Nonetheless, the RAM model seems to be fairly sound, and has done a good job of modelling typical machine technology since the early 60's.

1.8 Example: 2-dimension maxima

Let us do an example that illustrates how we analyze algorithms. Suppose you want to buy a car. You want the pick the fastest car. But fast cars are expensive; you want the cheapest. You cannot decide which is more important: speed or price. Definitely do not want a car if there is another that is both faster and cheaper. We say that the fast, cheap car *dominates* the slow, expensive car relative to your selection criteria. So, given a collection of cars, we want to list those cars that are not dominated by any other.

Here is how we might model this as a formal problem.

- Let a point p in 2-dimensional space be given by its integer coordinates, $p = (p.x, p.y)$.

- A point p is said to be dominated by point q if $p.x \leq q.x$ and $p.y \leq q.y$.
- Given a set of n points, $P = \{p_1, p_2, \ldots, p_n\}$ in 2-space a point is said to be maximal if it is not dominated by any other point in P.

The car selection problem can be modelled this way: For each car we associate (x, y) pair where x is the speed of the car and y is the negation of the price. High y value means a cheap car and low y means expensive car. Think of y as the money left in your pocket after you have paid for the car. Maximal points correspond to the fastest and cheapest cars.

The *2-dimensional Maxima* is thus defined as

- Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in 2-space, output the set of maximal points of P, i.e., those points p_i such that p_i is not dominated by any other point of P.

Here is set of maximal points for a given set of points in 2-d.

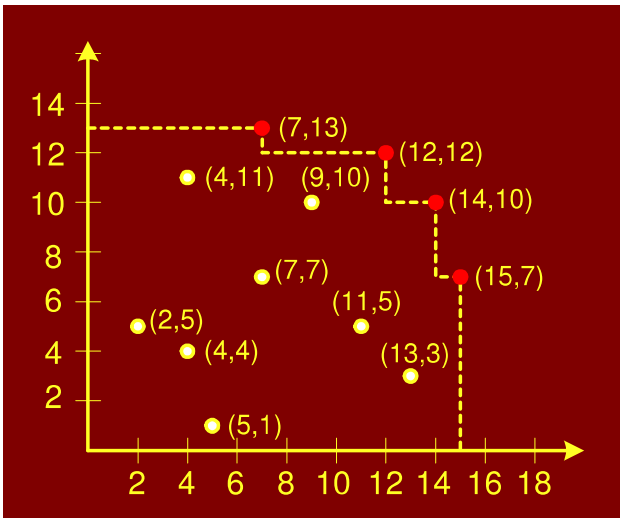

Figure 1.1: Maximal points in 2-d

Our description of the problem is at a fairly mathematical level. We have intentionally not discussed how the points are represented. We have not discussed any input or output formats. We have avoided programming and other software issues.

1.9 Brute-Force Algorithm

To get the ball rolling, let's just consider a simple brute-force algorithm, with no thought to efficiency. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the initial set of points. For each point p_i, test it against all other points p_j. If p_i is not dominated by any other point, then output it.

This English description is clear enough that any (competent) programmer should be able to implement it. However, if you want to be a bit more formal, it could be written in pseudocode as follows:

```
MAXIMA(int n, Point P[1...n])
1  for i ← 1 to n
2  do maximal ← true
3     for j ← 1 to n
4     do
5        if (i ≠ j) and (P[i].x ≤ P[j].x) and (P[i].y ≤ P[j].y)
6           then maximal ← false;  break
7     if (maximal = true)
8       then output P[i]
```

There are no formal rules to the syntax of this pseudo code. In particular, do not assume that more detail is better. For example, I omitted type specifications for the procedure Maxima and the variable maximal, and I never defined what a Point data type is, since I felt that these are pretty clear from context or just unimportant details. Of course, the appropriate level of detail is a judgement call. Remember, algorithms are to be read by people, and so the level of detail depends on your intended audience. When writing pseudo code, you should omit details that detract from the main ideas of the algorithm, and just go with the essentials.

You might also notice that I did not insert any checking for consistency. For example, I assumed that the points in P are all distinct. If there is a duplicate point then the algorithm may fail to output even a single point. (Can you see why?) Again, these are important considerations for implementation, but we will often omit error checking because we want to see the algorithm in its simplest form.

Here are a series of figures that illustrate point domination.

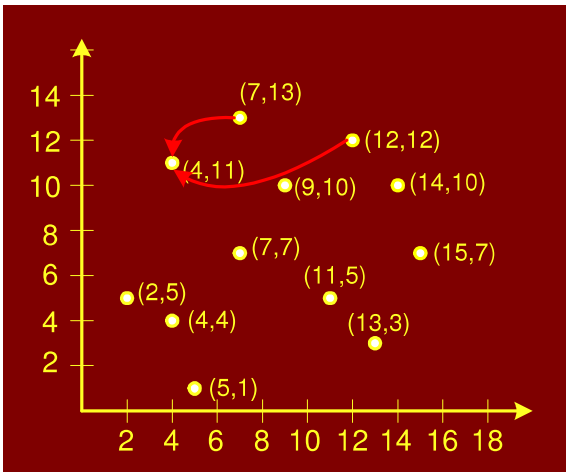

Figure 1.2: Points that dominate $(4, 11)$

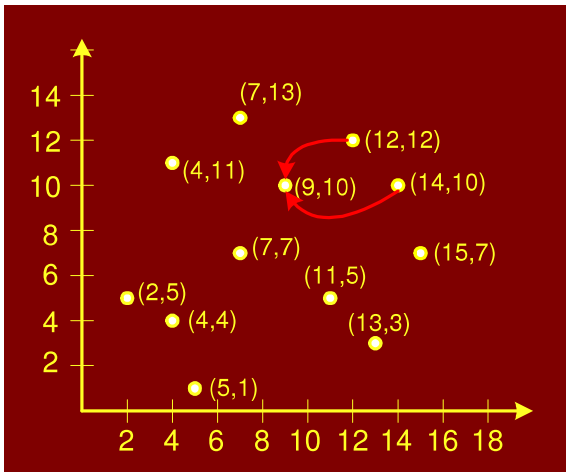

Figure 1.3: Points that dominate $(9, 10)$

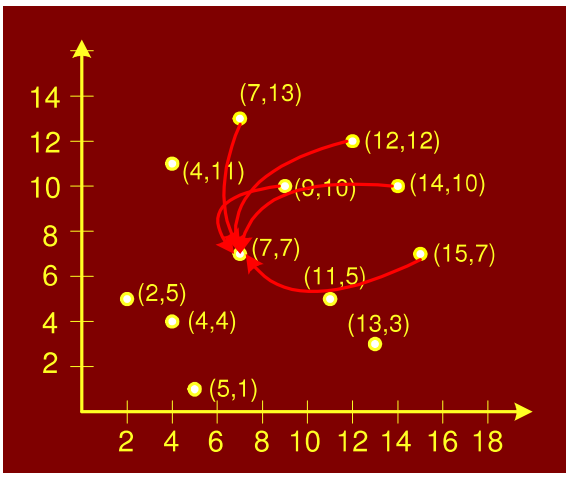

Figure 1.4: Points that dominate $(7,7)$

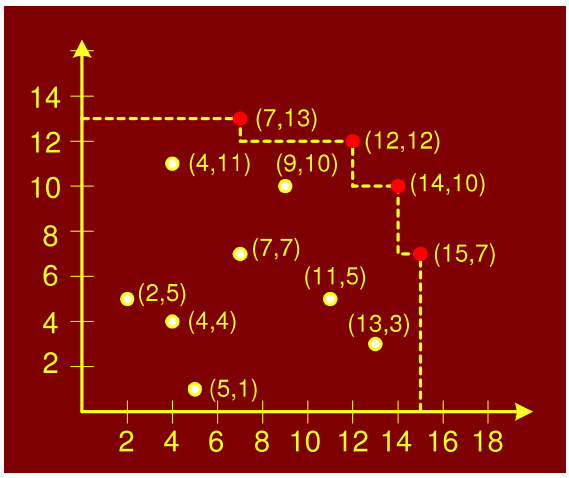

Figure 1.5: The maximal points

1.10 Running Time Analysis

The main purpose of our mathematical analysis will be measuring the execution time. We will also be concerned about the space (memory) required by the algorithm.

The running time of an implementation of the algorithm would depend upon the speed of the computer, programming language, optimization by the compiler etc. Although important, we will ignore these technological issues in our analysis.

To measure the running time of the brute-force 2-d maxima algorithm, we could count the number of steps of the pseudo code that are executed or, count the number of times an element of P is accessed or, the number of comparisons that are performed.

The running time depends upon the input size, e.g. n Different inputs of the same size may result in different running time. For example, breaking out of the inner loop in the brute-force algorithm depends not only on the input size of P but also the structure of the input.

Two criteria for measuring running time are *worst-case time* and *average-case time*.

Worst-case time is the maximum running time over all (legal) inputs of size n. Let I denote an input instance, let $|I|$ denote its length, and let $T(I)$ denote the running time of the algorithm on input I. Then

$$T_{worst}(n) = \max_{|I|=n} T(I)$$

Average-case time is the average running time over all inputs of size n. Let $p(I)$ denote the probability of seeing this input. The average-case time is the weighted sum of running times with weights

being the probabilities:

$$T_{avg}(n) = \sum_{|I|=n} p(I)T(I)$$

We will almost always work with worst-case time. Average-case time is more difficult to compute; it is difficult to specify probability distribution on inputs. Worst-case time will specify an upper limit on the running time.

1.10.1 Analysis of the brute-force maxima algorithm.

Assume that the input size is n, and for the running time we will count the number of time that any element of P is accessed. Clearly we go through the outer loop n times, and for each time through this loop, we go through the inner loop n times as well. The condition in the if-statement makes four accesses to P. The output statement makes two accesses for each point that is output. In the worst case every point is maximal (can you see how to generate such an example?) so these two access are made for each time through the outer loop.

```
MAXIMA(int n, Point P[1...n])
1  for i ← 1 to n                    n times
2  do maximal ← true
3     for j ← 1 to n                 n times
4     do
5        if (i ≠ j)&(P[i].x ≤ P[j].x)&(P[i].y ≤ P[j].y)   4 accesses
6          then maximal ← false break
7     if maximal
8       then output P[i].x, P[i].y    2 accesses
```

Thus we might express the worst-case running time as a pair of nested summations, one for the i-loop and the other for the j-loop:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} \left(2 + \sum_{j=1}^{n} 4\right) \\ &= \sum_{i=1}^{n} (4n + 2) \\ &= (4n + 2)n = 4n^2 + 2n \end{aligned}$$

For small values of n, any algorithm is fast enough. What happens when n gets large? Running time does become an issue. When n is large, n^2 term will be much larger than the n term and will *dominate* the running time.

We will say that the worst-case running time is $\Theta(n^2)$. This is called the asymptotic growth rate of the function. We will discuss this Θ-notation more formally later.

The analysis involved computing a summation. Summation should be familiar but let us review a bit here. Given a finite sequence of values $a_1, a_2, \ldots, a_n$, their sum $a_1 + a_2 + \ldots + a_n$ is expressed in summation notation as

$$\sum_{i=1}^{n} a_i$$

If $n = 0$, then the sum is additive identity, 0.

Some facts about summation: If c is a constant

$$\sum_{i=1}^{n} c a_i = c \sum_{i=1}^{n} a_i$$

and

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

Some important summations that should be committed to memory.

Arithmetic series

$$\begin{aligned} \sum_{i=1}^{n} i &= 1 + 2 + \ldots + n \\ &= \frac{n(n+1)}{2} = \Theta(n^2) \end{aligned}$$

Quadratic series

$$\begin{aligned} \sum_{i=1}^{n} i^2 &= 1 + 4 + 9 + \ldots + n^2 \\ &= \frac{2n^3 + 3n^2 + n}{6} = \Theta(n^3) \end{aligned}$$

Geometric series

$$\begin{aligned} \sum_{i=1}^{n} x^i &= 1 + x + x^2 + \ldots + x^n \\ &= \frac{x^{(n+1)} - 1}{x - 1} = \Theta(n^2) \end{aligned}$$

If $0 < x < 1$ then this is $\Theta(1)$, and if $x > 1$, then this is $\Theta(x^n)$.

Harmonic series For $n \geq 0$

$$\begin{aligned} H_n &= \sum_{i=1}^{n} \frac{1}{i} \\ &= 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n \\ &= \Theta(\ln n) \end{aligned}$$

1.11 Analysis: A Harder Example

Let us consider a harder example.

```
NESTED-LOOPS()
1  for i ← 1 to n
2  do
3      for j ← 1 to 2i
4      do k = j ...
5          while (k ≥ 0)
6          do k = k − 1 ...
```

How do we analyze the running time of an algorithm that has complex nested loop? The answer is we write out the loops as summations and then solve the summations. To convert loops into summations, we work from inside-out.

Consider the *inner most while* loop.

```
NESTED-LOOPS()
1  for i ← 1 to n
2  do for j ← 1 to 2i
3      do k = j
4          while (k ≥ 0)
5          do k = k − 1
```

It is executed for $k = j, j-1, j-2, \ldots, 0$. Time spent inside the while loop is constant. Let $I()$ be the time spent in the while loop. Thus

$$I(j) = \sum_{k=0}^{j} 1 = j + 1$$

Consider the *middle for* loop.

```
NESTED-LOOPS()
1  for i ← 1 to n
2  do for j ← 1 to 2i
3      do k = j
4          while (k ≥ 0)
5          do k = k − 1
```

Its running time is determined by i. Let $M()$ be the time spent in the for loop:

$$\begin{aligned} M(i) &= \sum_{j=1}^{2i} I(j) \\ &= \sum_{j=1}^{2i} (j+1) \\ &= \sum_{j=1}^{2i} j + \sum_{j=1}^{2i} 1 \\ &= \frac{2i(2i+1)}{2} + 2i \\ &= 2i^2 + 3i \end{aligned}$$

Finally the *outer-most for* loop.

```
NESTED-LOOPS()
1  for i ← 1 to n
2  do for j ← 1 to 2i
3       do k = j
4          while (k ≥ 0)
5          do k = k − 1
```

Let $T()$ be running time of the entire algorithm:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} M(i) \\ &= \sum_{i=1}^{n} (2i^2 + 3i) \\ &= \sum_{i=1}^{n} 2i^2 + \sum_{i=1}^{n} 3i \\ &= 2\frac{2n^3 + 3n^2 + n}{6} + 3\frac{n(n+1)}{2} \\ &= \frac{4n^3 + 15n^2 + 11n}{6} \\ &= \Theta(n^3) \end{aligned}$$

1.11.1 2-dimension Maxima Revisited

Recall the 2-d maxima problem: Let a point p in 2-dimensional space be given by its integer coordinates, $p = (p.x, p.y)$. A point p is said to *dominated* by point q if $p.x \leq q.x$ and $p.y \leq q.y$. Given a set of n

points, $P = \{p_1, p_2, \ldots, p_n\}$ in 2-space a point is said to be *maximal* if it is not dominated by any other point in P. The problem is to output all the maximal points of P. We introduced a brute-force algorithm that ran in $\Theta(n^2)$ time. It operated by comparing *all pairs* of points. Is there an approach that is significantly better?

The problem with the brute-force algorithm is that it uses no intelligence in *pruning* out decisions. For example, once we know that a point p_i is dominated by another point p_j, we do not need to use p_i for eliminating other points. This follows from the fact that dominance relation is *transitive*. If p_j dominates p_i and p_i dominates p_h then p_j also dominates p_h; p_i is not needed.

1.11.2 Plane-sweep Algorithm

The question is whether we can make an significant improvement in the running time? Here is an idea for how we might do it. We will sweep a vertical line across the plane from left to right. As we sweep this line, we will build a structure holding the maximal points lying to the left of the sweep line. When the sweep line reaches the rightmost point of P , then we will have constructed the complete set of maxima. This approach of solving geometric problems by sweeping a line across the plane is called *plane sweep*.

Although we would like to think of this as a continuous process, we need some way to perform the plane sweep in discrete steps. To do this, we will begin by sorting the points in increasing order of their x-coordinates. For simplicity, let us assume that no two points have the same y-coordinate. (This limiting assumption is actually easy to overcome, but it is good to work with the simpler version, and save the messy details for the actual implementation.) Then we will advance the sweep-line from point to point in n discrete steps. As we encounter each new point, we will update the current list of maximal points.

We will sweep a vertical line across the 2-d plane from left to right. As we sweep, we will build a structure holding the maximal points lying to the left of the sweep line. When the sweep line reaches the rightmost point of P, we will have the complete set of maximal points. We will store the existing maximal points in a list The points that p_i dominates will appear at the end of the list because points are sorted by x-coordinate. We will scan the list left to right. Every maximal point with y-coordinate less than p_i will be eliminated from computation. We will add maximal points onto the end of a list and delete from the end of the list. We can thus use a stack to store the maximal points. The point at the top of the stack will have the highest x-coordinate.

Here are a series of figures that illustrate the plane sweep. The figure also show the content of the stack.

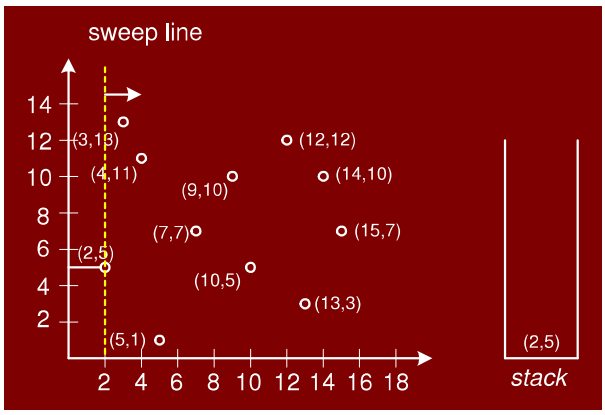

Figure 1.6: Sweep line at $(2, 5)$

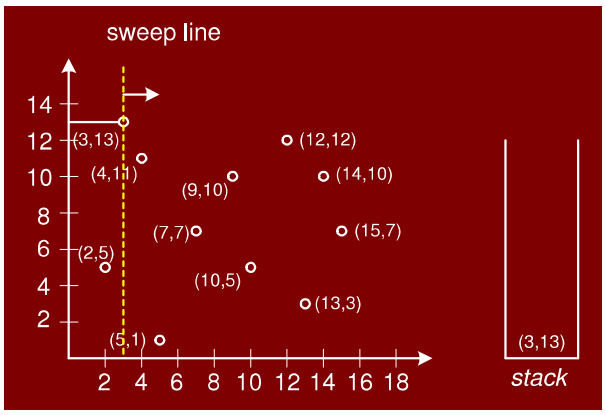

Figure 1.7: Sweep line at $(3, 13)$

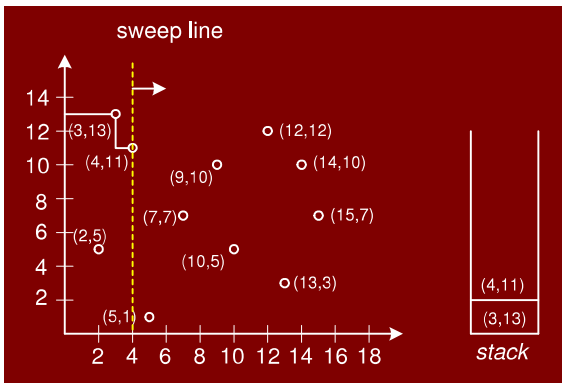

Figure 1.8: Sweep line at $(4, 11)$

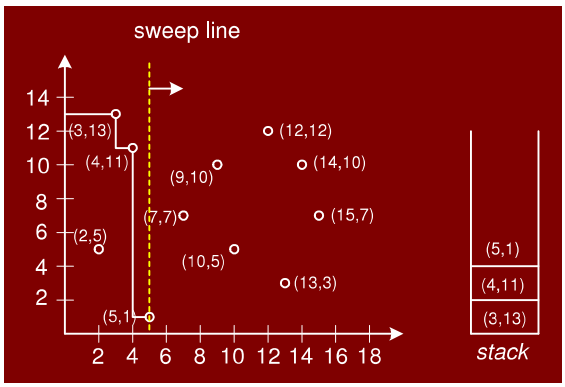

Figure 1.9: Sweep line at $(5, 1)$

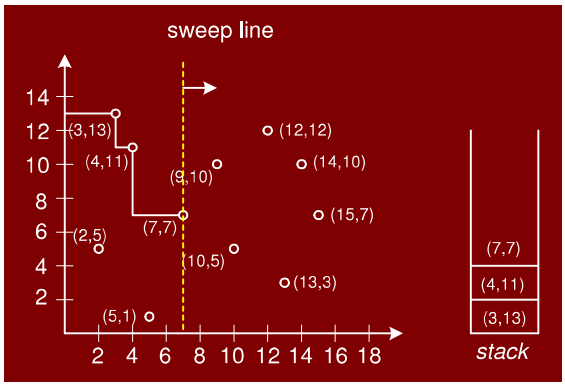

Figure 1.10: Sweep line at $(7, 7)$

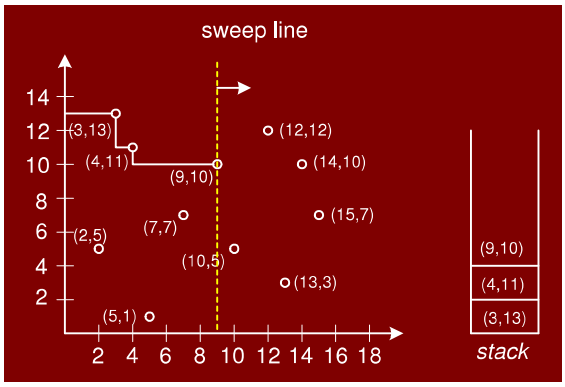

Figure 1.11: Sweep line at $(9, 10)$

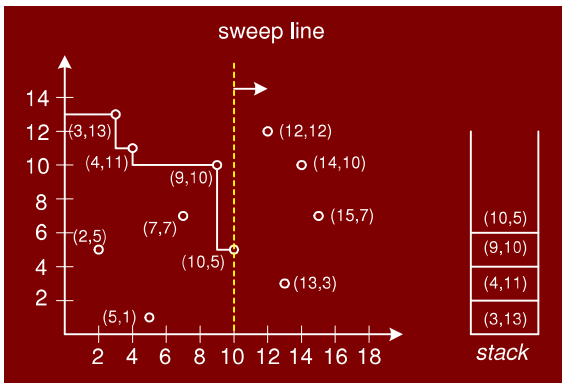

Figure 1.12: Sweep line at $(10, 5)$

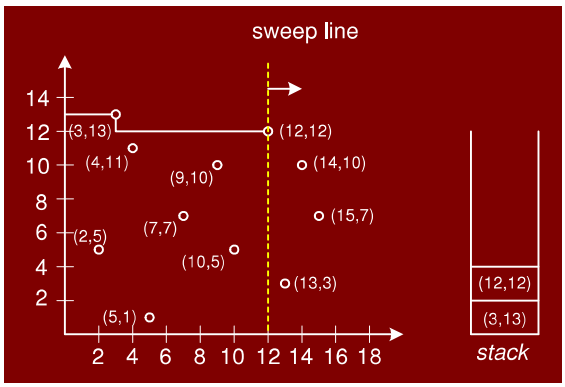

Figure 1.13: Sweep line at $(12, 12)$

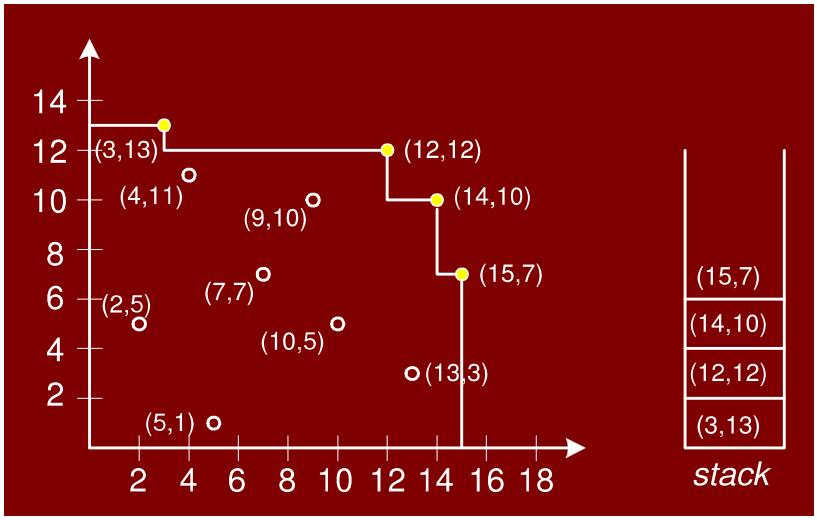

Figure 1.14: The final maximal set

Here is the algorithm.

PLANE-SWEEP-MAXIMA$(n,\ P[1..n])$

```
1  sort P in increasing order by x;
2  stack s;
3  for i ← 1 to n
4  do
5      while (s.notEmpty() & s.top().y ≤ P[i].y)
6      do s.pop();
7      s.push(P[i]);
8  output the contents of stack s;
```

1.11.3 Analysis of Plane-sweep Algorithm

Sorting takes $\Theta(n \log n)$; we will show this later when we discuss sorting. The for loop executes n times. The inner loop (seemingly) could be iterated $(n-1)$ times. It seems we still have an $n(n-1)$ or $\Theta(n^2)$ algorithm. Got fooled by simple minded loop-counting. The while loop will not execute more n times over the entire course of the algorithm. Why is this? Observe that the total number of elements that can be pushed on the stack is n since we execute exactly one push each time during the outer for-loop.

We pop an element off the stack each time we go through the inner while-loop. It is impossible to pop more elements than are ever pushed on the stack. Therefore, the inner while-loop cannot execute more than n times over the entire course of the algorithm. (*Make sure that you understand this*).

The for-loop iterates n times and the inner while-loop also iterates n time for a total of $\Theta(n)$. Combined with the sorting, the runtime of entire plane-sweep algorithm is $\Theta(n \log n)$.

1.11.4 Comparison of Brute-force and Plane sweep algorithms

How much of an improvement is plane-sweep over brute-force? Consider the ratio of running times:

$$\frac{n^2}{n\log n} = \frac{n}{\log n}$$

n	$\log n$	$\frac{n}{\log n}$
100	7	15
1000	10	100
10000	13	752
100000	17	6021
1000000	20	50171

For $n = 1,000,000$, if plane-sweep takes 1 second, the brute-force will take about 14 hours!. From this we get an idea about the importance of asymptotic analysis. It tells us which algorithm is better for large values of n. As we mentioned before, if n is not very large, then almost any algorithm will be fast. But efficient algorithm design is most important for large inputs, and the general rule of computing is that input sizes continue to grow until people can no longer tolerate the running times. Thus, by designing algorithms efficiently, you make it possible for the user to run large inputs in a reasonable amount of time.