```
1.  Let $v_0$ be the corner of $C_0$.
2.  Let $h_1, \dots, h_n$ be the half-planes of $H$.
3.  for $i \leftarrow 1$ to $n$
4.      do if $v_{i-1} \in h_i$
5.           then $v_i \leftarrow v_{i-1}$
6.           else $v_i \leftarrow$ the point $p$ on $\ell_i$ that maximizes $f_{\vec{c}}(p)$, subject to the
                  constraints in $H_{i-1}$.
7.                if $p$ does not exist
8.                  then Report that the linear program is infeasible and quit.
9.  return $v_n$
```

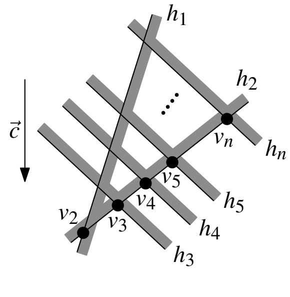
We now analyze the performance of our algorithm.

Lemma 4.7 *Algorithm* 2DBOUNDEDLP *computes the solution to a bounded linear program with n constraints and two variables in $O(n^2)$ time and linear storage.*

Proof. To prove that the algorithm correctly finds the solution, we have to show that after every stage—whenever we have added a new half-plane h_i—the point v_i is still the optimum point for C_i. This follows immediately from Lemma 4.5. If the 1-dimensional linear program on ℓ_i is infeasible, then C_i is empty, and consequently $C = C_n \subseteq C_i$ is empty, which means that the linear program is infeasible.

It is easy to see that the algorithm requires only linear storage. We add the half-planes one by one in n stages. The time spent in stage i is dominated by the time to solve a 1-dimensional linear program in line 6, which is $O(i)$. Hence, the total time needed is bounded by

$$\sum_{i=1}^{n} O(i) = O(n^2). \qquad \square$$



Although our linear programming algorithm is nice and simple, its running time is disappointing—the algorithm is much slower than the previous algorithm, which computed the whole feasible region. Is our analysis too crude? We bounded the cost of every stage i by $O(i)$. This is not always a tight bound: Stage i takes $\Theta(i)$ time only when $v_{i-1} \notin h_i$; when $v_{i-1} \in h_i$ then stage i takes constant time. So if we could bound the number of times the optimal vertex changes, we might be able to prove a better running time. Unfortunately the optimum vertex can change n times: there are orders for some configurations where every new half-plane makes the previous optimum illegal. The figure in the margin shows such an example. This means that the algorithm will really spend $\Theta(n^2)$ time. How can we avoid this nasty situation?

4.4 Randomized Linear Programming

If we have a second look at the example where the optimum changes n times, we see that the problem is not so much that the set of half-planes is bad. If we had added them in the order h_n, h_{n-1}, ..., h_3, then the optimal vertex would not change anymore after the addition of h_n. In this case the running time would be $O(n)$. Is this a general phenomenon? Is it true that, for any set H of half-planes, there is a good order to treat them? The answer to this question is "yes," but that doesn't seem to help us much. Even if such a good order exists, there seems to be no easy way to actually *find* it. Remember that we have to find the order at the beginning of the algorithm, when we don't know anything about the intersection of the half-planes yet.

We now meet a quite intriguing phenomenon. Although we have no way to determine an ordering of H that is guaranteed to lead to a good running time, we have a very simple way out of our problem. We simply pick a *random* ordering of H. Of course, we could have bad luck and pick an order that leads to a quadratic running time. But with some luck, we pick an order that makes it run much faster. Indeed, we shall prove below that most orders lead to a fast algorithm. For completeness, we first repeat the algorithm.

```
Algorithm 2dRandomizedBoundedLP($H, \vec{c}, m_1, m_2$)
Input. A linear program ($H \cup \{m_1, m_2\}, \vec{c}$), where $H$ is a set of $n$ half-planes,
   $\vec{c} \in \mathbb{R}^2$, and $m_1$, $m_2$ bound the solution.
Output. If ($H \cup \{m_1, m_2\}, \vec{c}$) is infeasible, then this fact is reported. Otherwise,
   the lexicographically smallest point $p$ that maximizes $f_{\vec{c}}(p)$ is reported.
1.  Let $v_0$ be the corner of $C_0$.
2.  Compute a random permutation $h_1, \dots, h_n$ of the half-planes by calling
    RandomPermutation($H[1 \cdots n]$).
3.  for $i \leftarrow 1$ to $n$
4.      do if $v_{i-1} \in h_i$
5.            then $v_i \leftarrow v_{i-1}$
6.            else $v_i \leftarrow$the point $p$ on $\ell_i$ that maximizes $f_{\vec{c}}(p)$, subject to the
                 constraints in $H_{i-1}$.
7.                 if $p$ does not exist
8.                   then Report that the linear program is infeasible and quit.
9.  return $v_n$
```

The only difference from the previous algorithm is in line 2, where we put the half-planes in random order before we start adding them one by one. To be able to do this, we assume that we have a random number generator, RANDOM(k), which has an integer k as input and generates a random integer between 1 and k in constant time. Computing a random permutation can then be done with the following linear time algorithm.

```
Algorithm RandomPermutation($A$)
Input. An array $A[1 \cdots n]$.
Output. The array $A[1 \cdots n]$ with the same elements, but rearranged into a
   random permutation.
1.  for $k \leftarrow n$ downto 2
2.      do rndindex $\leftarrow$Random($k$)
3.         Exchange $A[k]$ and $A[rndindex]$.
```

The new linear programming algorithm is called a *randomized algorithm*; its running time depends on certain random choices made by the algorithm. (In the linear programming algorithm, these random choices were made in the subroutine RANDOMPERMUTATION.)

What is the running time of this randomized version of our incremental linear programming algorithm? There is no easy answer to that. It all depends on the order that is computed in line 2. Consider a fixed set H of n half-planes. 2DRANDOMIZEDBOUNDEDLP treats them depending on the permutation chosen in line 2. Since there are $n!$ possible permutations of n objects, there are $n!$ possible ways in which the algorithm can proceed, each with its own running time. Because the permutation is random, each of these running times is equally likely. So what we do is analyze the *expected running time* of the algorithm, which is the *average running time over all $n!$ possible permutations*. The lemma below states that the expected running time of our randomized linear programming algorithm is $O(n)$. It is important to realize that we do not make any assumptions about the input: the expectancy is with respect to the random order in which the half-planes are treated and holds for any set of half-planes.

Lemma 4.8 *The 2-dimensional linear programming problem with n constraints can be solved in $O(n)$ randomized expected time using worst-case linear storage.*

Proof. As we observed before, the storage needed by the algorithm is linear.

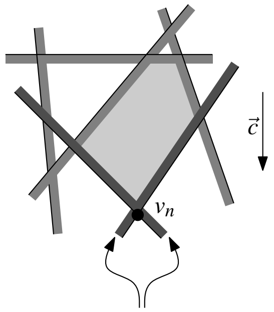
The running time RANDOMPERMUTATION is $O(n)$, so what remains is to analyze the time needed to add the half-planes $h_1, \dots, h_n$. Adding a half-plane takes constant time when the optimal vertex does not change. When the optimal vertex does change we need to solve a 1-dimensional linear program. We now bound the time needed for all these 1-dimensional linear programs.

Let X_i be a random variable, which is 1 if $v_{i-1} \notin h_i$, and 0 otherwise. Recall that a 1-dimensional linear program on i constraints can be solved in $O(i)$ time. The total time spent in line 6 over all half-planes $h_1, \dots, h_n$ is therefore

$$\sum_{i=1}^{n} O(i) \cdot X_i.$$

To bound the expected value of this sum we will use *linearity of expectation*: the expected value of a sum of random variables is the sum of the expected values of the random variables. This holds even if the random variables are dependent. Hence, the expected time for solving all 1-dimensional linear programs is

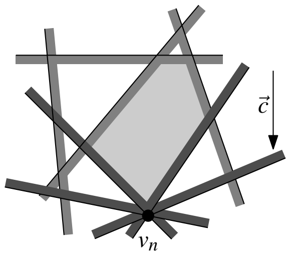
$$\mathrm{E}[\sum_{i=1}^{n} O(i) \cdot X_i] = \sum_{i=1}^{n} O(i) \cdot \mathrm{E}[X_i].$$



half-planes defining v_n

But what is $\mathrm{E}[X_i]$? It is exactly the probability that $v_{i-1} \notin h_i$. Let's analyze this probability.

We will do this with a technique called *backwards analysis*: we look at the algorithm "backwards." Assume that it has already finished, and that it has computed the optimum vertex v_n. Since v_n is a vertex of C_n, it is defined by at least two of the half-planes. Now we make one step backwards in time, and look at C_{n-1}. Note that C_{n-1} is obtained from C_n by removing the half-plane h_n. When does the optimum point change? This happens exactly if v_n is not a vertex of C_{n-1} that is extreme in the direction $\vec{c}$, which is only possible if h_n is one of the half-planes that define v_n. But the half-planes are added in random order, so h_n is a random element of $\{h_1, h_2, \ldots, h_n\}$. Hence, the probability that h_n is one of the half-planes defining v_n is at most $2/n$. Why do we say "at most"? First, it is possible that the boundaries of more than two half-planes pass through v_n. In that case, removing one of the two half-planes containing the edges incident to v_n may fail to change v_n. Furthermore, v_n may be defined by m_1 or m_2, which are not among the n candidates for the random choice of h_n. In both cases the probability is less than $2/n$.



The same argument works in general: to bound $\mathrm{E}[X_i]$, we fix the subset of the first i half-planes. This determines C_i. To analyze what happened in the last step, when we added h_i, we think backwards. The probability that we had to compute a new optimal vertex when adding h_i is the same as the probability that the optimal vertex changes when we remove a half-plane from C_i. The latter event only takes place for at most two half-planes of our fixed set $\{h_1, \ldots, h_i\}$. Since the half-planes are added in random order, the probability that h_i is one of the special half-planes is at most $2/i$. We derived this probability under the condition that the first i half-planes are some fixed subset of H. But since the derived bound holds for *any* fixed subset, it holds unconditionally. Hence, $\mathrm{E}[X_i] \leqslant 2/i$. We can now bound the expected total time for solving all 1-dimensional linear programs by

$$\sum_{i=1}^{n} O(i) \cdot \frac{2}{i} = O(n).$$

We already noted that the time spent in the rest of the algorithm is $O(n)$ as well. $\square$

Note again that the expectancy here is solely with respect to the random choices made by the algorithm. We do not average over possible choices for the input. For *any* input set of n half-planes, the expected running time of the algorithm is $O(n)$; there are no bad inputs.

4.5 Unbounded Linear Programs

In the preceding sections we avoided handling the case of an unbounded linear program by adding two additional, artificial constraints. This is not always a suitable solution. Even if the linear program is bounded, we may not know a large enough bound M. Furthermore, unbounded linear programs do occur in practice, and we have to solve them correctly.

Let's first see how we can recognize whether a given linear program $(H, \vec{c})$ is unbounded. As we saw before, that means that there is a ray ρ completely