Computational Geometry

Chapter 4

Linear Programming

On the Agenda

- ❑ Linear programming
- ❑ Duality
- ❑ Smallest enclosing disk

Define:

i – types of foods (0< *i* <d+1).
j – types of vitamins (1≤*j*≤n).
x_i – the amount of food of type *i.*
a_{ji} – the amount of vitamin *j* in one unit of food *i.*
c_i – the number of calories in one unit of food *i.*
b_j – minimal required amount of vitamin *j.*

Constraints (we need to consume some minimal amount of each vitamin):

Want to minimize the cost, subject to :

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1d}x_d \geq b_1$$
$$\vdots$$
$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nd}x_d \geq b_n$$

$$C(x) = c_1x_1 + c_2x_2 + \cdots + c_dx_d$$

Define:

i – types of foods (0< *i* <d+1).
j – types of vitamins (1≤*j*≤n).
x_i – the amount of food of type *i.*
a_{ji} – the amount of vitamin *j* in one unit of food *i.*
c_i – the number of calories in one unit of food *i.*
b_j – minimal required amount of vitamin *j.*

Constraints (we need to consume some minimal amount of each vitamin):

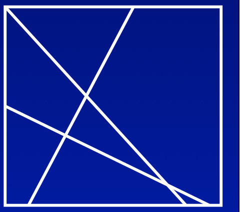
Want to minimize the cost, subject to :

$$\begin{aligned} &Minimize: \ c^T x \\ &Subject\ to: \ Ax \geq b \end{aligned}$$

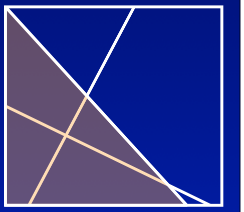
$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1d}x_d \geq b_1$$
$$\vdots$$
$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nd}x_d \geq b_n$$

$$C(x) = c_1x_1 + c_2x_2 + \cdots + c_dx_d$$

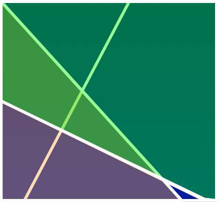
Linear Programming – The Geometry



Linear Programming – The Geometry

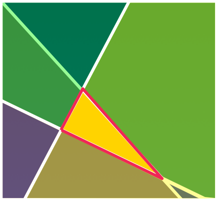


Linear Programming – The Geometry



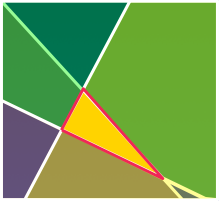
Linear Programming – The Geometry

Linear Programming – The Geometry



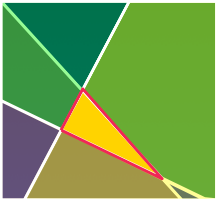
Linear Programming – The Geometry

- Each constraint defines defines a half-space region in *d*-dimensional space.



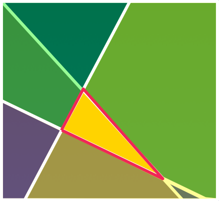
Linear Programming – The Geometry

- Each constraint defines defines a half-space region in *d*-dimensional space.
- The *feasible region* is the (convex) intersection of these half-spaces.



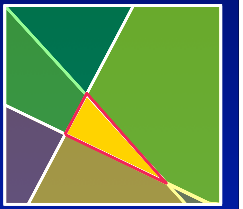
Linear Programming – The Geometry

- Each constraint defines defines a half-space region in *d*-dimensional space.
- The *feasible region* is the (convex) intersection of these half-spaces.



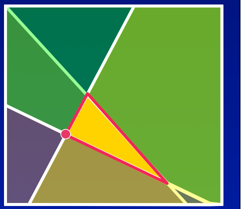
Linear Programming – The Geometry

- Each constraint defines defines a half-space region in *d*-dimensional space.
- The *feasible region* is the (convex) intersection of these half-spaces.
- We will treat the case *d* = 2, where each constraint defines a *half-plane*.

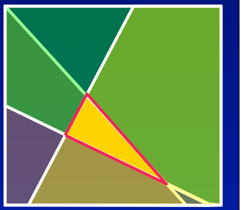


Linear Programming – The Geometry

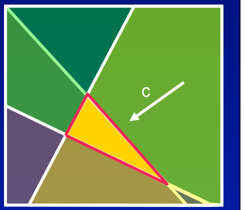
- Each constraint defines defines a half-space region in *d*-dimensional space.
- The *feasible region* is the (convex) intersection of these half-spaces.
- We will treat the case *d* = 2, where each constraint defines a *half-plane*.



More Geometry

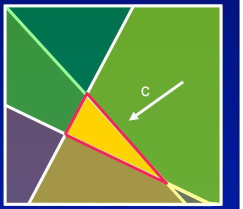


More Geometry



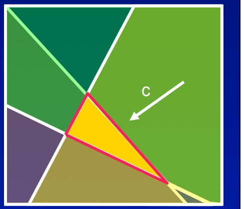
More Geometry

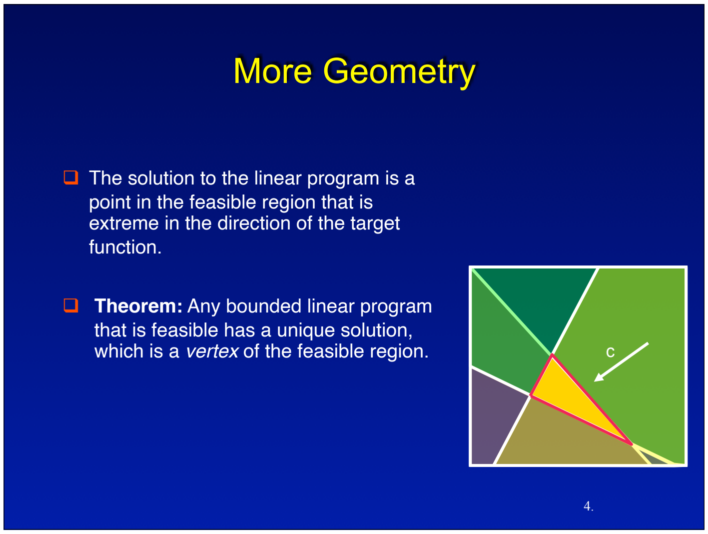
- The solution to the linear program is a point in the feasible region that is extreme in the direction of the target function.



More Geometry

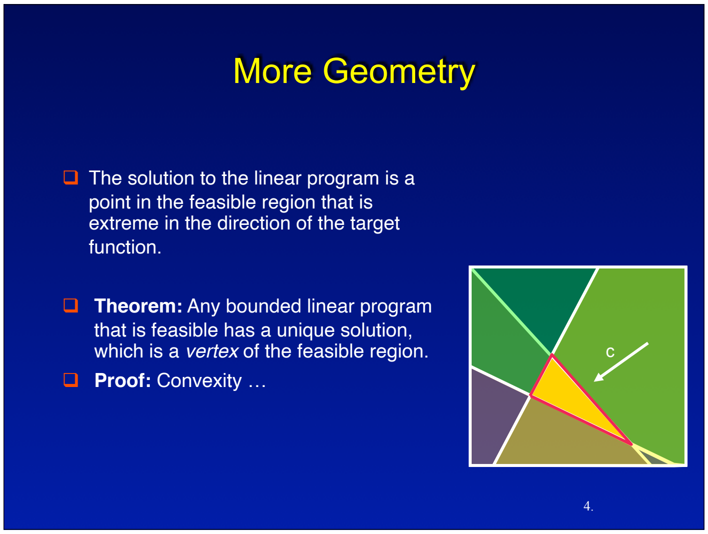
- The solution to the linear program is a point in the feasible region that is extreme in the direction of the target function.





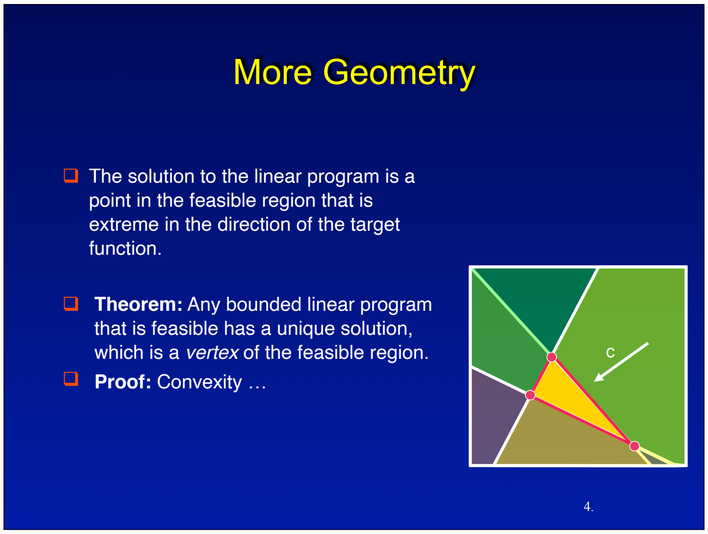
More Geometry

- The solution to the linear program is a point in the feasible region that is extreme in the direction of the target function.
- **Theorem:** Any bounded linear program that is feasible has a unique solution, which is a *vertex* of the feasible region.



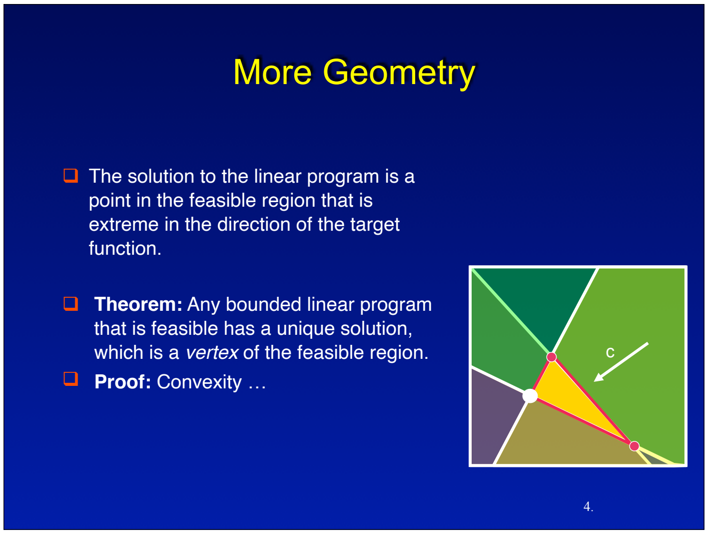
More Geometry

- The solution to the linear program is a point in the feasible region that is extreme in the direction of the target function.
- **Theorem:** Any bounded linear program that is feasible has a unique solution, which is a *vertex* of the feasible region.
- **Proof:** Convexity …



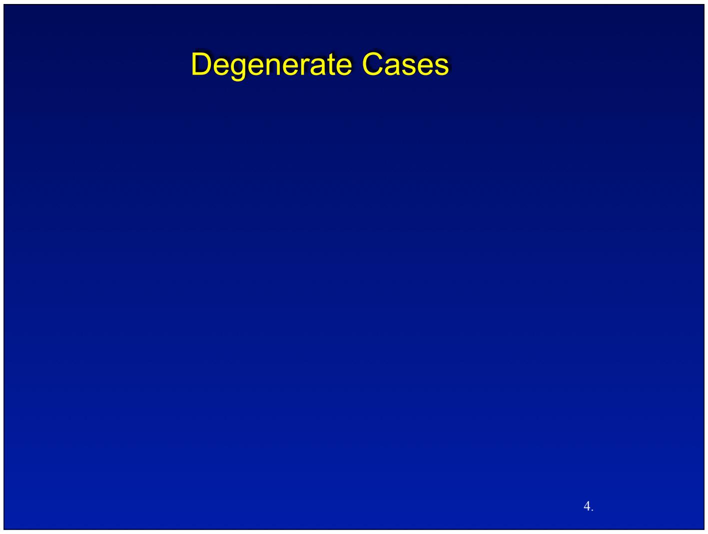
More Geometry

- The solution to the linear program is a point in the feasible region that is extreme in the direction of the target function.
- **Theorem:** Any bounded linear program that is feasible has a unique solution, which is a *vertex* of the feasible region.
- **Proof:** Convexity …

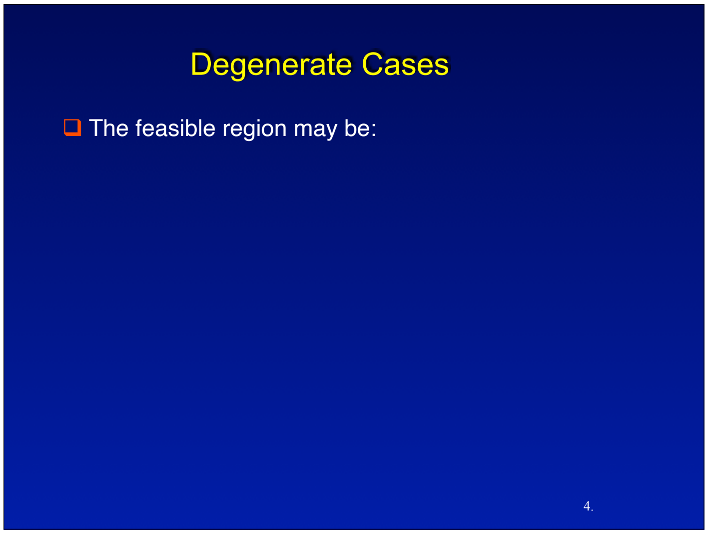


More Geometry

- The solution to the linear program is a point in the feasible region that is extreme in the direction of the target function.
- **Theorem:** Any bounded linear program that is feasible has a unique solution, which is a *vertex* of the feasible region.
- **Proof:** Convexity …



Degenerate Cases



Degenerate Cases

- The feasible region may be:

Degenerate Cases

- The feasible region may be:

Degenerate Cases

- The feasible region may be:

Degenerate Cases

- The feasible region may be:

Degenerate Cases

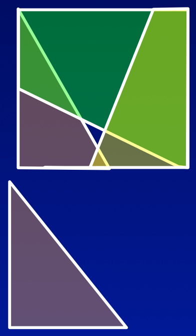
- The feasible region may be:
  - Empty

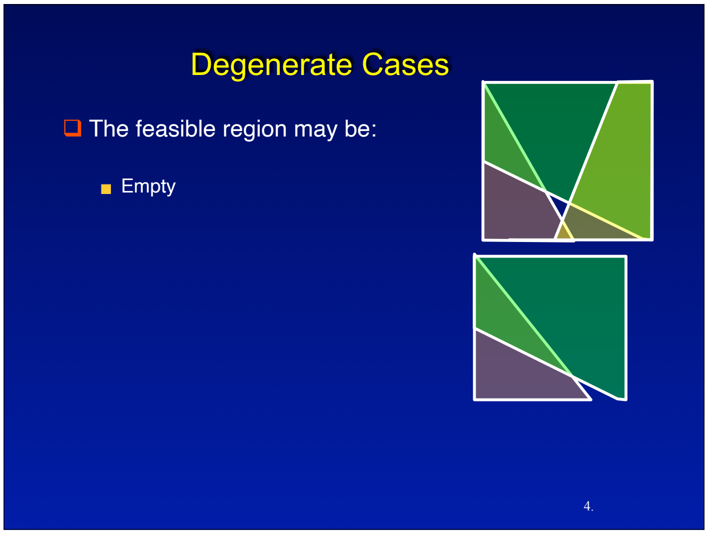
Degenerate Cases

- The feasible region may be:
  - Empty

Degenerate Cases

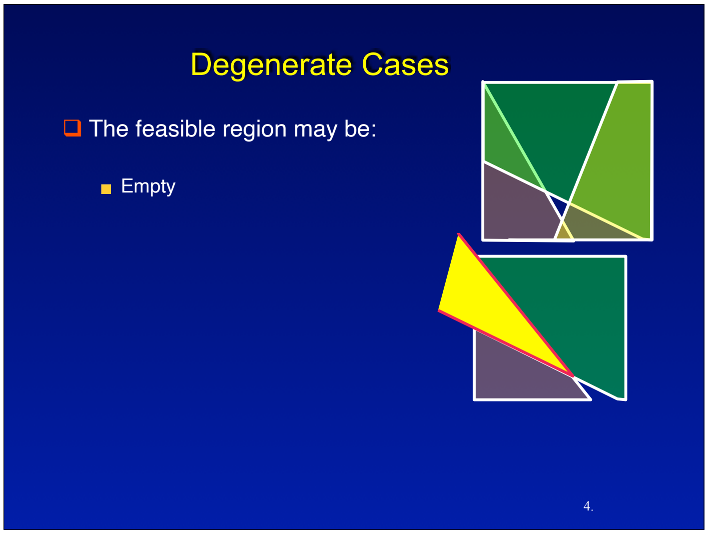
- The feasible region may be:
  - Empty





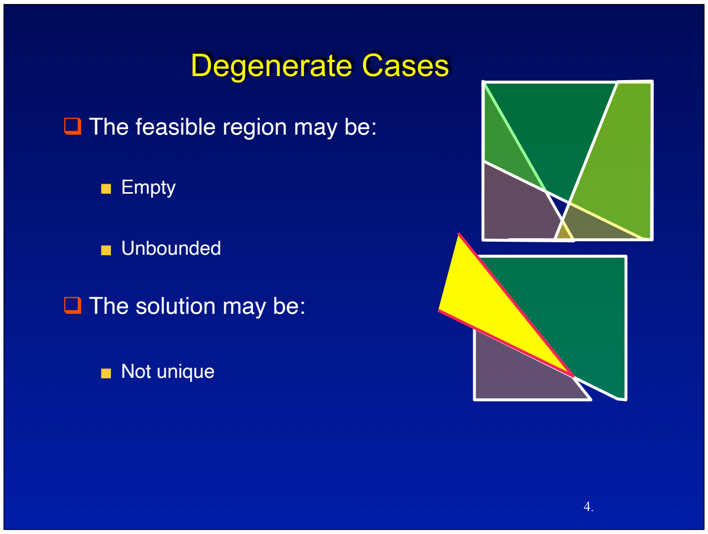
Degenerate Cases

- The feasible region may be:
  - Empty



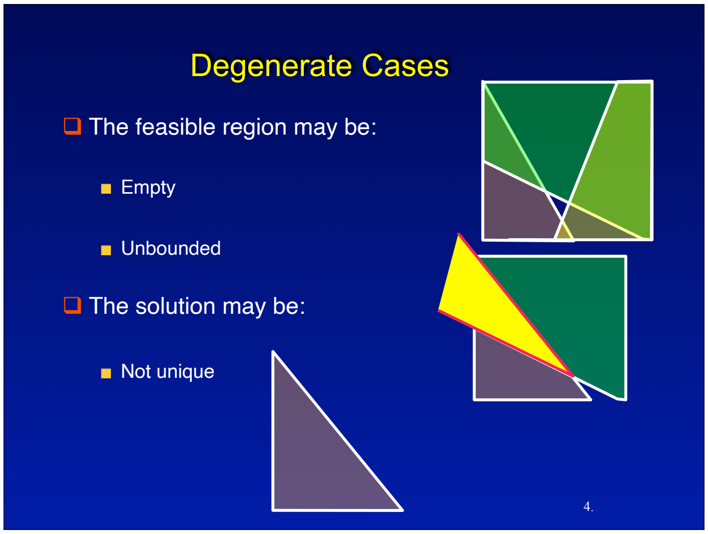
Degenerate Cases

- The feasible region may be:
  - Empty



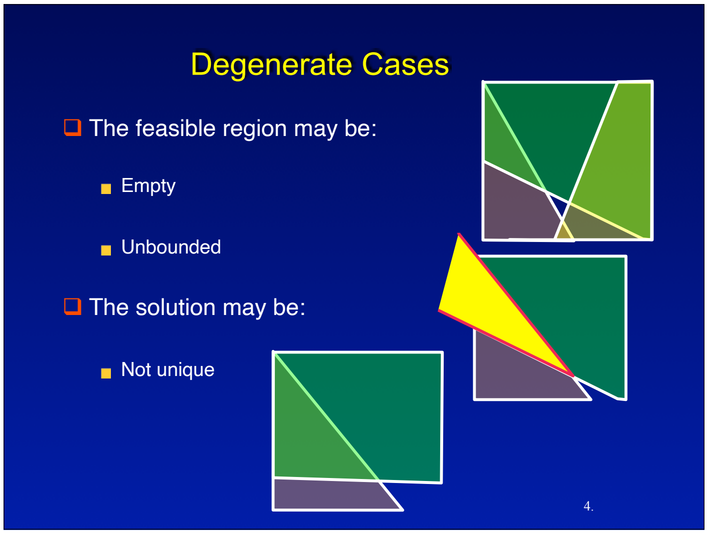
Degenerate Cases

- The feasible region may be:
  - Empty
  - Unbounded
- The solution may be:
  - Not unique



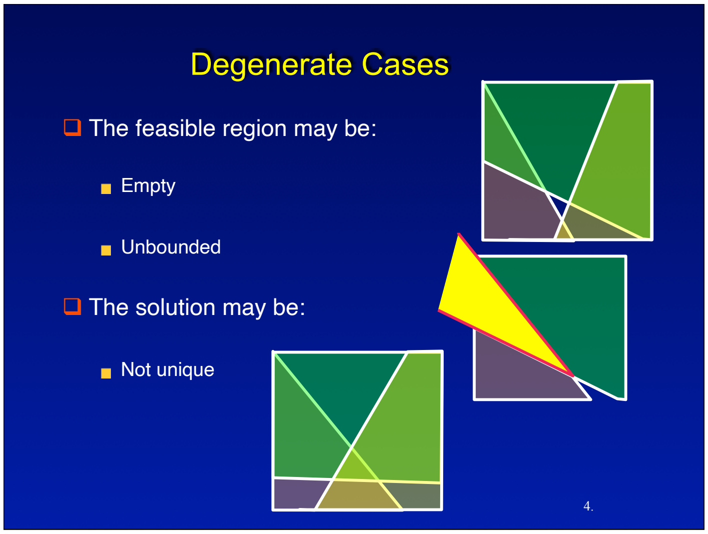
Degenerate Cases

- The feasible region may be:
  - Empty
  - Unbounded
- The solution may be:
  - Not unique



Degenerate Cases

- The feasible region may be:
  - Empty
  - Unbounded
- The solution may be:
  - Not unique

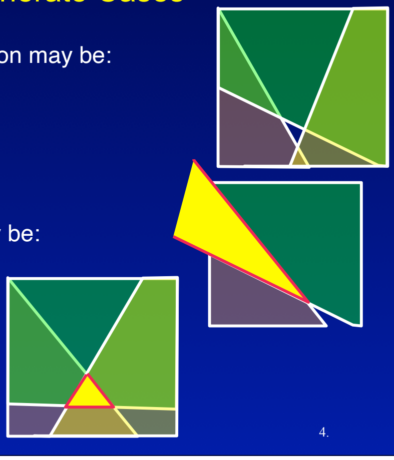


Degenerate Cases

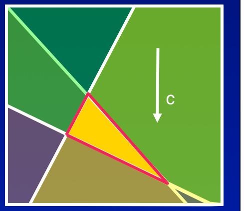
- The feasible region may be:
  - Empty
  - Unbounded
- The solution may be:
  - Not unique

Degenerate Cases

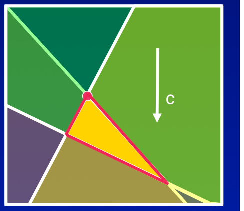
- The feasible region may be:
  - Empty
  - Unbounded
- The solution may be:
  - Not unique



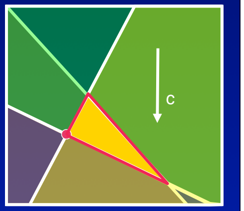
The Simplex Algorithm



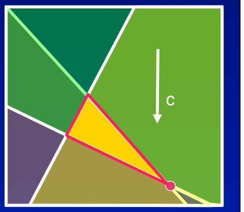
The Simplex Algorithm



The Simplex Algorithm

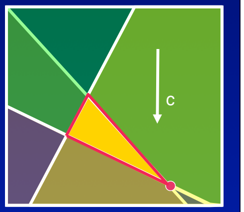


The Simplex Algorithm



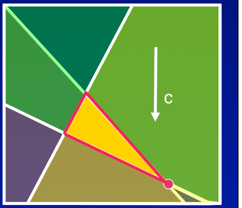
The Simplex Algorithm

- Assume WLOG that the cost function points “downwards”.



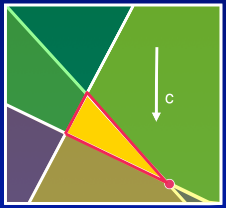
The Simplex Algorithm

- Assume WLOG that the cost function points “downwards”.
- Construct (some of) the vertices of the feasible region.



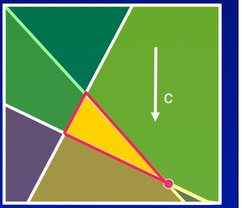
The Simplex Algorithm

- Assume WLOG that the cost function points “downwards”.
- Construct (some of) the vertices of the feasible region.
- Walk edge by edge downwards until reaching a local minimum (which is also a global minimum).



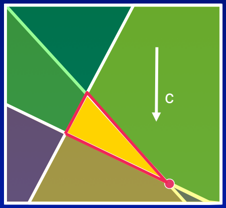
The Simplex Algorithm

- Assume WLOG that the cost function points “downwards”.
- Construct (some of) the vertices of the feasible region.
- Walk edge by edge downwards until reaching a local minimum (which is also a global minimum).



The Simplex Algorithm

- Assume WLOG that the cost function points “downwards”.
- Construct (some of) the vertices of the feasible region.
- Walk edge by edge downwards until reaching a local minimum (which is also a global minimum).
- In R^d, the number of vertices might be $\Theta(n^{\lfloor d/2 \rfloor})$.



LP History

- Mid 20th century: Simplex algorithm, time complexity $\Theta(n^{\lfloor d/2 \rfloor})$ in the **worst** case.
- 1980’s (Khachiyan) ellipsoid algorithm with time complexity poly(n,d).
- 1980’s (Karmakar) interior-point algorithm with time complexity poly(n,d).
- 1984 (Megiddo) – parametric search algorithm with time complexity $O(C_d n)$ where C_d is a constant dependent only on d. E.g. $C_d = 2^{d^2}$.
- The holy grail: An algorithm with complexity independent of d.

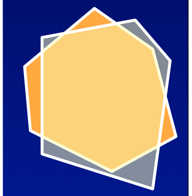
- In practice the simplex algorithm is used because of its linear *expected* runtime.

$O(n \log n)$ 2D Linear Programming

- Input:
  - n half planes.
  - Cost function that WLOG “points down”.
- Algorithm:
  1. Partition the n half-planes into two groups.
  2. Compute, recursively, the feasible region for each group.
  3. Compute the intersection of the two feasible regions.
  4. Check the cost function on the region vertices.

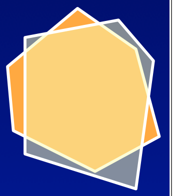
Divide and Conquer – Complexity

- Stage 3:
  - Intersection of two convex polygons – plane sweep algorithm.
  - No more than four segments are ever in the SLS and no more than eight events in the EQ – O(n).
- Stage 4:
  - Find the minimal cost vertex - O(n).



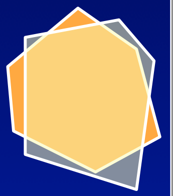
Divide and Conquer – Complexity

- Stage 3:
  - Intersection of two convex polygons – plane sweep algorithm.
  - No more than four segments are ever in the SLS and no more than eight events in the EQ – O(n).
- Stage 4:
  - Find the minimal cost vertex - O(n).



Divide and Conquer – Complexity

- Stage 3:
  - Intersection of two convex polygons – plane sweep algorithm.
  - No more than four segments are ever in the SLS and no more than eight events in the EQ – O(n).
- Stage 4:
  - Find the minimal cost vertex - O(n).

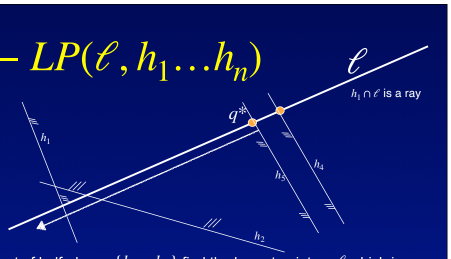


$$T(n) = 2T(n/2)+O(n) \Rightarrow$$
$$T(n) = O(n \log n)$$

O(n^2) Incremental Algorithm

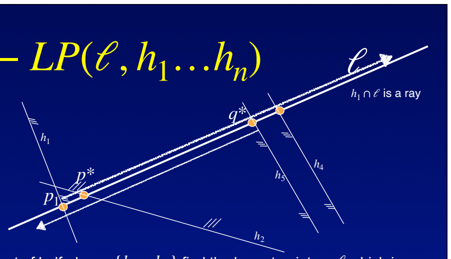
- The idea:
  - Start by intersecting two halfplanes.
  - Add halfplanes one by one and update optimal vertex by solving one-dimensional LP problem on new line *if needed*.

$1D - LP(\ell, h_1 \ldots h_n)$

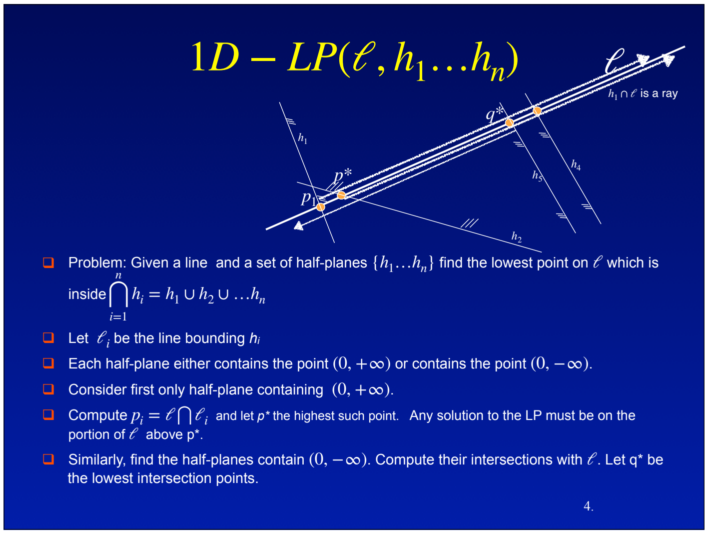


- Problem: Given a line and a set of half-planes $\{h_1 \ldots h_n\}$ find the lowest point on ℓ which is inside $\bigcap_{i=1}^{n} h_i = h_1 \cup h_2 \cup \ldots h_n$
- Let ℓ_i be the line bounding h_i
- Each half-plane either contains the point $(0, +\infty)$ or contains the point $(0, -\infty)$.
- Consider first only half-plane containing $(0, +\infty)$.
- Compute $p_i = \ell \bigcap \ell_i$ and let p^* the highest such point. Any solution to the LP must be on the portion of ℓ above p*.
- Similarly, find the half-planes contain $(0, -\infty)$. Compute their intersections with ℓ. Let q* be the lowest intersection points.

$1D - LP(\ell, h_1 \ldots h_n)$

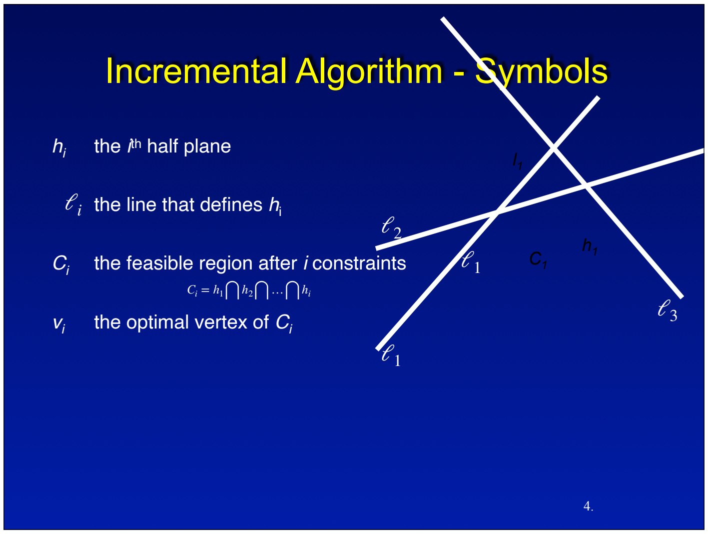


- Problem: Given a line and a set of half-planes $\{h_1 \ldots h_n\}$ find the lowest point on ℓ which is inside $\bigcap_{i=1}^{n} h_i = h_1 \cup h_2 \cup \ldots h_n$
- Let ℓ_i be the line bounding h_i
- Each half-plane either contains the point $(0, +\infty)$ or contains the point $(0, -\infty)$.
- Consider first only half-plane containing $(0, +\infty)$.
- Compute $p_i = \ell \bigcap \ell_i$ and let p^* the highest such point. Any solution to the LP must be on the portion of ℓ above p*.
- Similarly, find the half-planes contain $(0, -\infty)$. Compute their intersections with ℓ. Let q* be the lowest intersection points.



$1D - LP(\ell, h_1 \ldots h_n)$

- Problem: Given a line and a set of half-planes $\{h_1 \ldots h_n\}$ find the lowest point on ℓ which is inside $\bigcap_{i=1}^{n} h_i = h_1 \cup h_2 \cup \ldots h_n$
- Let ℓ_i be the line bounding h_i
- Each half-plane either contains the point $(0, +\infty)$ or contains the point $(0, -\infty)$.
- Consider first only half-plane containing $(0, +\infty)$.
- Compute $p_i = \ell \bigcap \ell_i$ and let p^* the highest such point. Any solution to the LP must be on the portion of ℓ above p*.
- Similarly, find the half-planes contain $(0, -\infty)$. Compute their intersections with ℓ. Let q* be the lowest intersection points.



Incremental Algorithm - Symbols

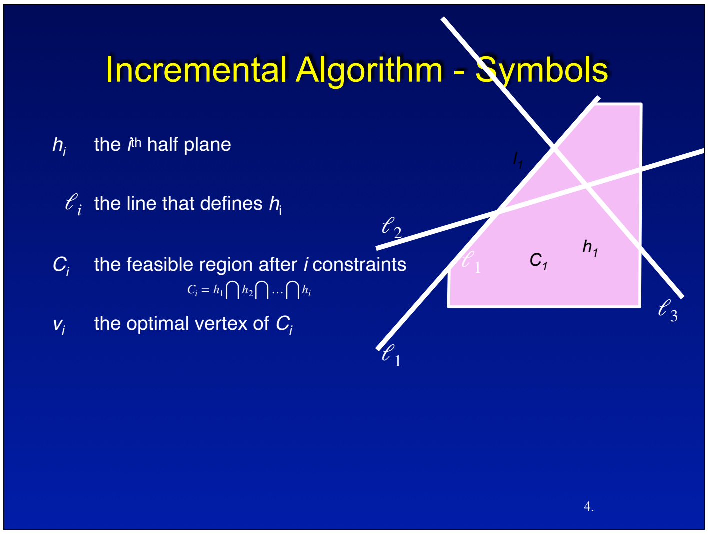
h_i the i^{th} half plane

ℓ_i the line that defines h_i

C_i the feasible region after i constraints

$C_i = h_1 \bigcap h_2 \bigcap \ldots \bigcap h_i$

v_i the optimal vertex of C_i



Incremental Algorithm - Symbols

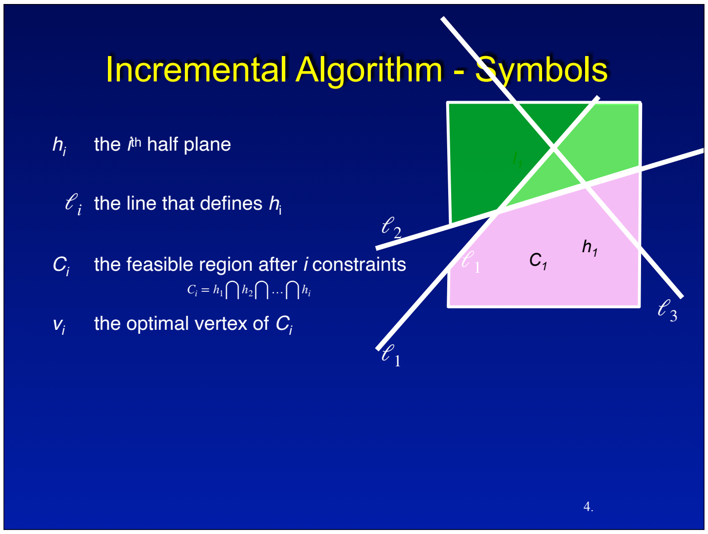
h_i the i^{th} half plane

ℓ_i the line that defines h_i

C_i the feasible region after i constraints

$C_i = h_1 \bigcap h_2 \bigcap \ldots \bigcap h_i$

v_i the optimal vertex of C_i



Incremental Algorithm - Symbols

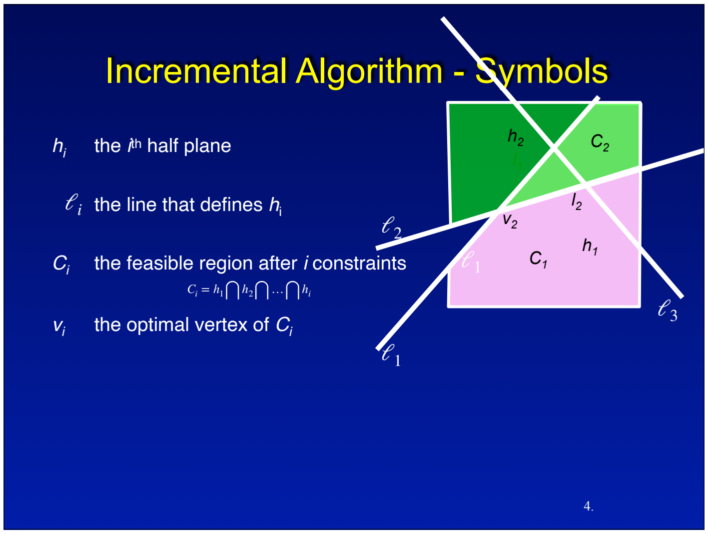
h_i the i^{th} half plane

ℓ_i the line that defines h_i

C_i the feasible region after i constraints

$C_i = h_1 \bigcap h_2 \bigcap \ldots \bigcap h_i$

v_i the optimal vertex of C_i



Incremental Algorithm - Symbols

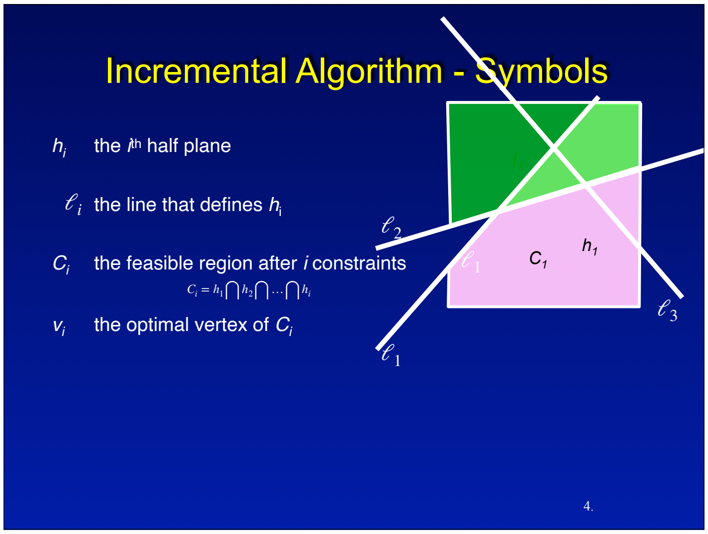
h_i the i^{th} half plane

ℓ_i the line that defines h_i

C_i the feasible region after i constraints

$C_i = h_1 \bigcap h_2 \bigcap \ldots \bigcap h_i$

v_i the optimal vertex of C_i



Incremental Algorithm - Symbols

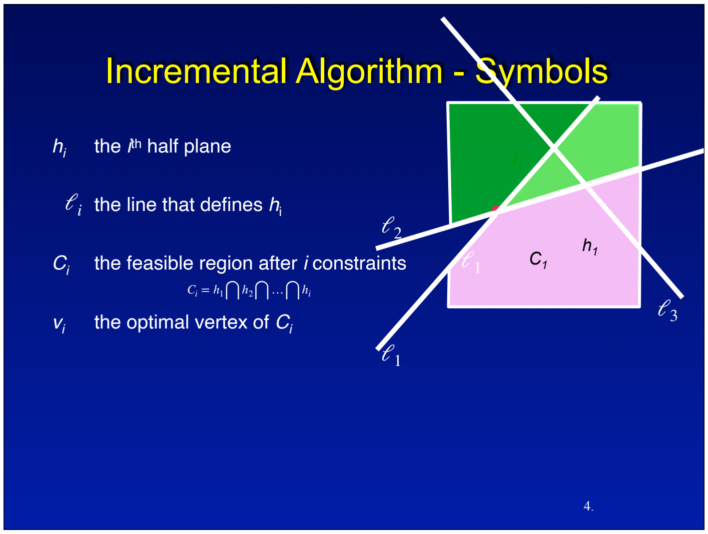
h_i the i^{th} half plane

ℓ_i the line that defines h_i

C_i the feasible region after i constraints

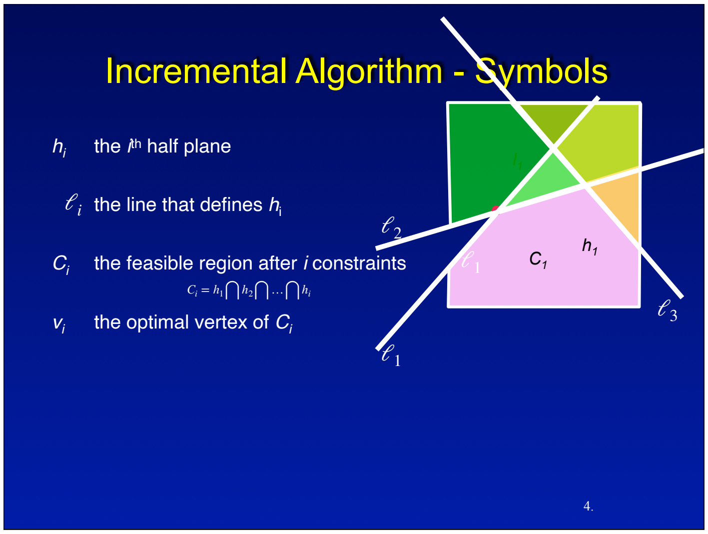
$C_i = h_1 \bigcap h_2 \bigcap \ldots \bigcap h_i$

v_i the optimal vertex of C_i



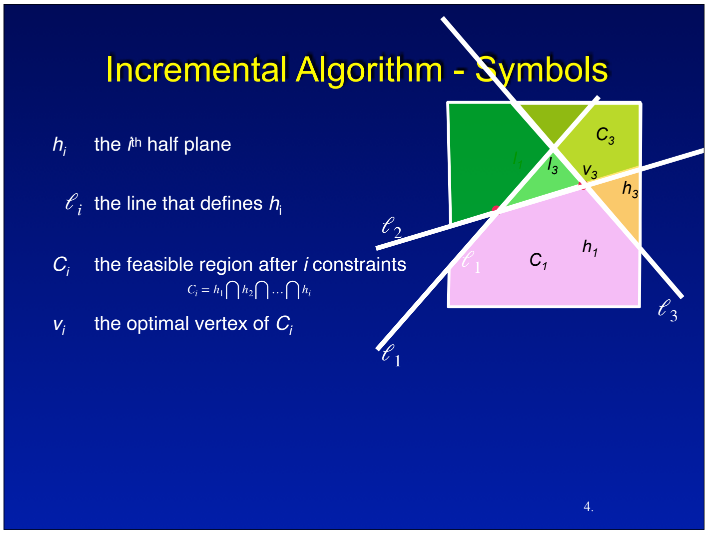
Incremental Algorithm - Symbols

h_i	the i^{th} half plane
ℓ_i	the line that defines h_i
C_i	the feasible region after i constraints $C_i = h_1 \bigcap h_2 \bigcap \dots \bigcap h_i$
v_i	the optimal vertex of C_i



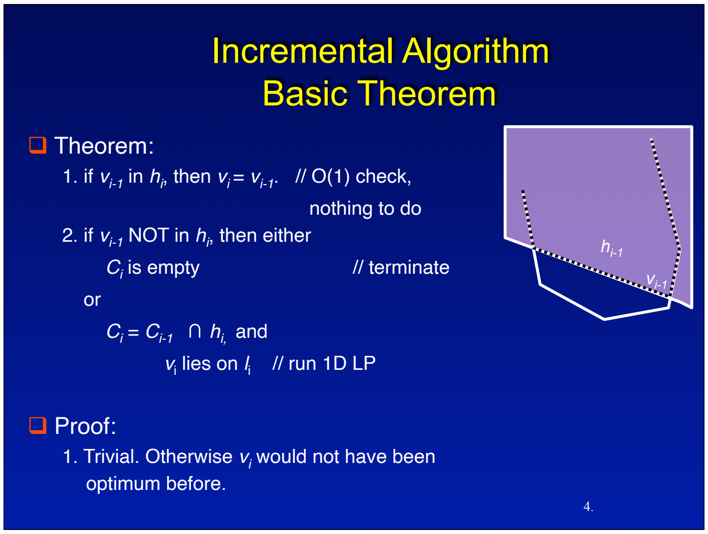
Incremental Algorithm - Symbols

h_i	the i^{th} half plane
ℓ_i	the line that defines h_i
C_i	the feasible region after i constraints $C_i = h_1 \bigcap h_2 \bigcap \dots \bigcap h_i$
v_i	the optimal vertex of C_i



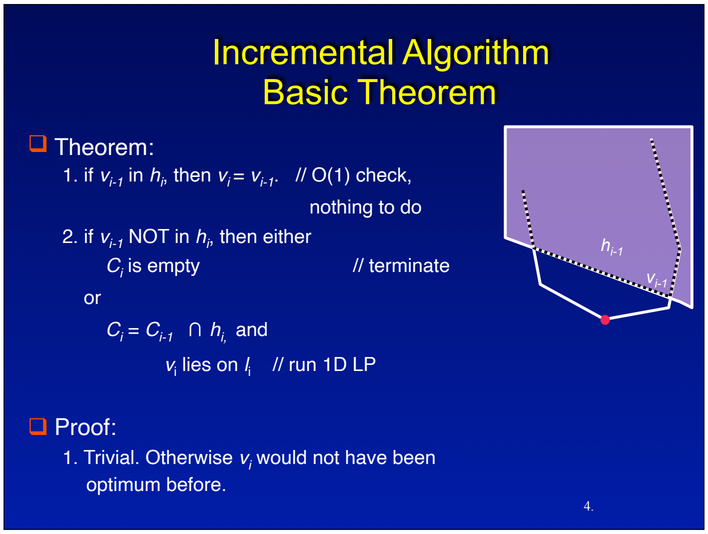
Incremental Algorithm - Symbols

h_i	the i^{th} half plane
ℓ_i	the line that defines h_i
C_i	the feasible region after i constraints $C_i = h_1 \bigcap h_2 \bigcap \dots \bigcap h_i$
v_i	the optimal vertex of C_i



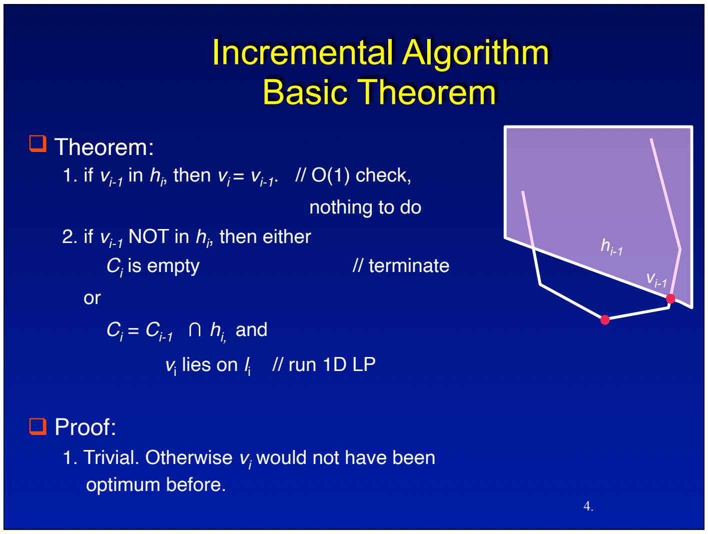
Incremental Algorithm
Basic Theorem

- Theorem:
  1. if v_{i-1} in h_i, then $v_i = v_{i-1}$. // O(1) check, nothing to do
  2. if v_{i-1} NOT in h_i, then either
     C_i is empty // terminate
     or
     $C_i = C_{i-1} \cap h_i$, and
     v_i lies on l_i // run 1D LP
- Proof:
  1. Trivial. Otherwise v_i would not have been optimum before.



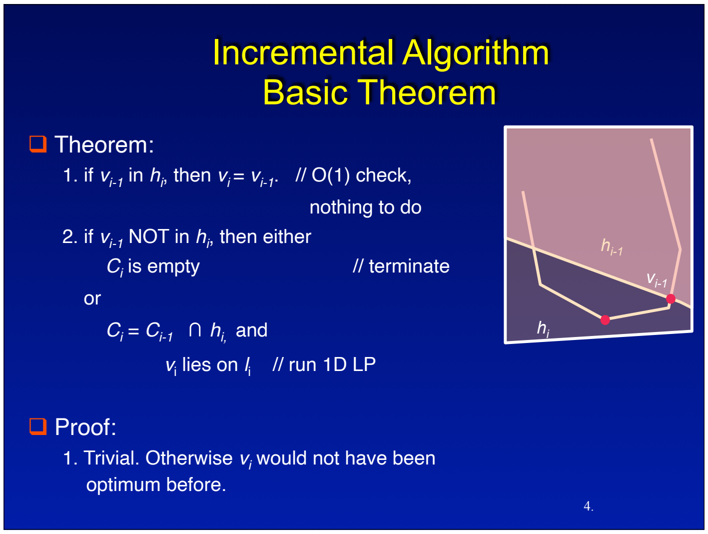
Incremental Algorithm
Basic Theorem

- Theorem:
  1. if v_{i-1} in h_i, then $v_i = v_{i-1}$. // O(1) check, nothing to do
  2. if v_{i-1} NOT in h_i, then either
     C_i is empty // terminate
     or
     $C_i = C_{i-1} \cap h_i$, and
     v_i lies on l_i // run 1D LP
- Proof:
  1. Trivial. Otherwise v_i would not have been optimum before.



Incremental Algorithm
Basic Theorem

- Theorem:
  1. if v_{i-1} in h_i, then $v_i = v_{i-1}$. // O(1) check, nothing to do
  2. if v_{i-1} NOT in h_i, then either
     C_i is empty // terminate
     or
     $C_i = C_{i-1} \cap h_i$, and
     v_i lies on l_i // run 1D LP
- Proof:
  1. Trivial. Otherwise v_i would not have been optimum before.



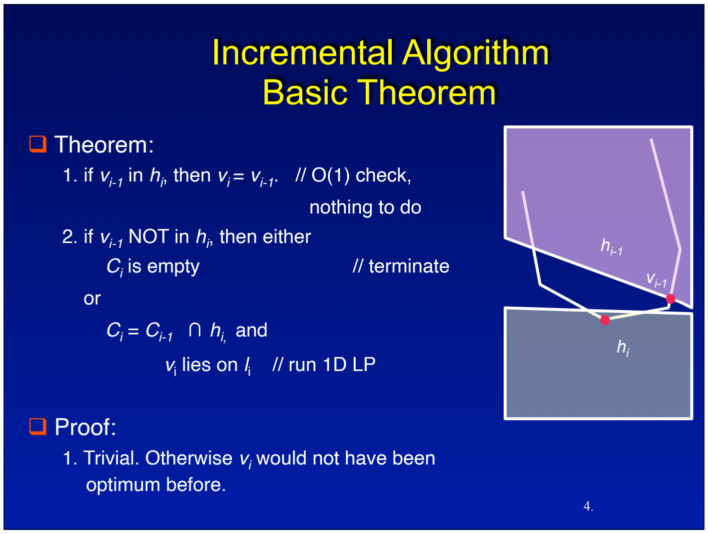
Incremental Algorithm
Basic Theorem

❑ Theorem:

1. if v_{i-1} in h_i, then $v_i = v_{i-1}$. // O(1) check, nothing to do
2. if v_{i-1} NOT in h_i, then either
   C_i is empty // terminate
   or
   $C_i = C_{i-1} \cap h_i$, and
   v_i lies on l_i // run 1D LP

❑ Proof:

1. Trivial. Otherwise v_i would not have been optimum before.



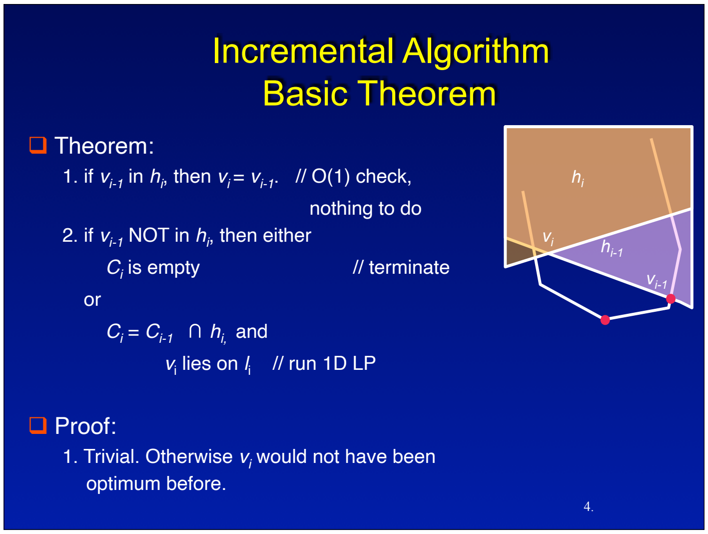
Incremental Algorithm
Basic Theorem

❑ Theorem:

1. if v_{i-1} in h_i, then $v_i = v_{i-1}$. // O(1) check, nothing to do
2. if v_{i-1} NOT in h_i, then either
   C_i is empty // terminate
   or
   $C_i = C_{i-1} \cap h_i$, and
   v_i lies on l_i // run 1D LP

❑ Proof:

1. Trivial. Otherwise v_i would not have been optimum before.



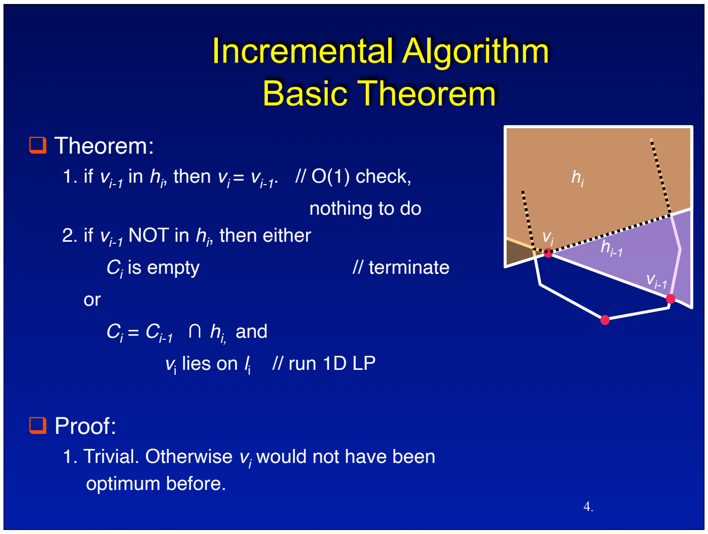
Incremental Algorithm
Basic Theorem

❑ Theorem:

1. if v_{i-1} in h_i, then $v_i = v_{i-1}$. // O(1) check, nothing to do
2. if v_{i-1} NOT in h_i, then either
   C_i is empty // terminate
   or
   $C_i = C_{i-1} \cap h_i$, and
   v_i lies on l_i // run 1D LP

❑ Proof:

1. Trivial. Otherwise v_i would not have been optimum before.



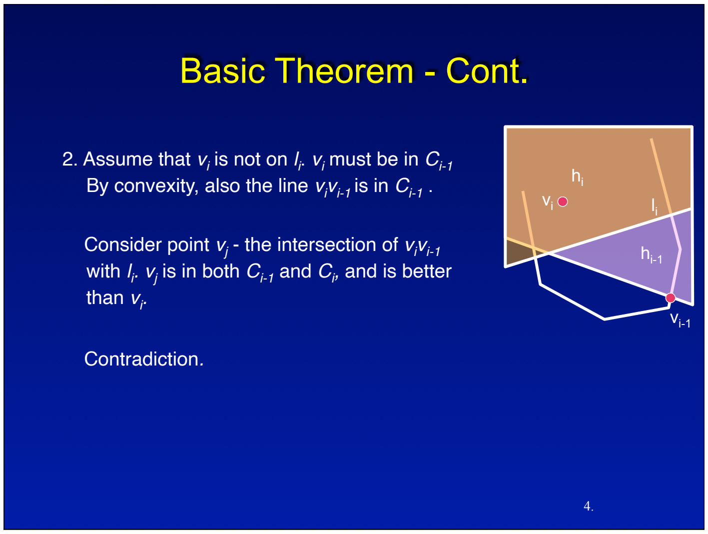
Incremental Algorithm
Basic Theorem

❑ Theorem:

1. if v_{i-1} in h_i, then $v_i = v_{i-1}$. // O(1) check, nothing to do
2. if v_{i-1} NOT in h_i, then either
   C_i is empty // terminate
   or
   $C_i = C_{i-1} \cap h_i$, and
   v_i lies on l_i // run 1D LP

❑ Proof:

1. Trivial. Otherwise v_i would not have been optimum before.

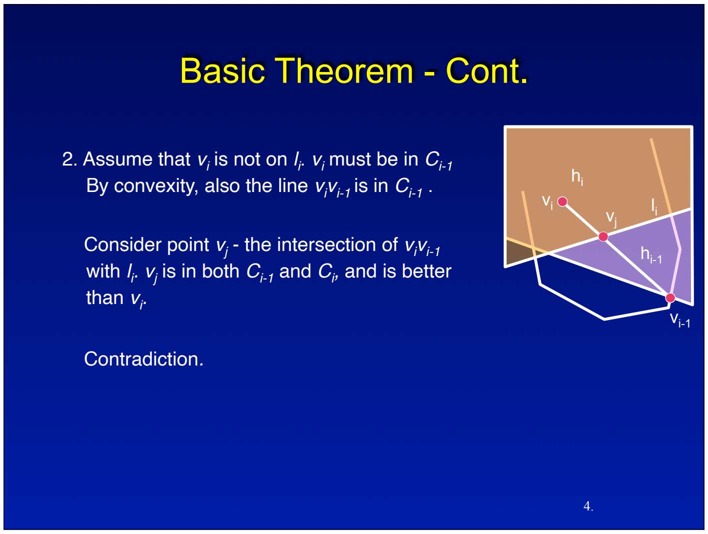


Basic Theorem - Cont.

2. Assume that v_i is not on l_i. v_i must be in C_{i-1}
   By convexity, also the line v_iv_{i-1} is in C_{i-1} .

   Consider point v_j - the intersection of v_iv_{i-1} with l_i. v_j is in both C_{i-1} and C_i, and is better than v_i.

   Contradiction.



Basic Theorem - Cont.

2. Assume that v_i is not on l_i. v_i must be in C_{i-1}
   By convexity, also the line v_iv_{i-1} is in C_{i-1} .

   Consider point v_j - the intersection of v_iv_{i-1} with l_i. v_j is in both C_{i-1} and C_i, and is better than v_i.

   Contradiction.

Finding v_i given l_i

(one-dimensional LP)

- Intersect each h_j ($j<i$) with l_i, generating i-1 rays representing (unbounded) intervals.
- Intersect the i-1 intervals in O(i) time.
- If the intersection is empty then report no solution, else report the lowest point.

Complexity Analysis

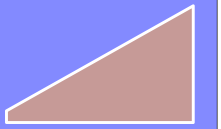
- $T(n) = \sum_{i=3}^{n} c \cdot i = c(3+4+5+\ldots n) = c \cdot n(n+1)/2 = \Theta(n^2)$

$$T(n) = \sum_{i=3}^{n} O(i) = O(n^2)$$

Complexity Analysis

- $T(n) = \sum_{i=3}^{n} c \cdot i = c(3+4+5+\ldots n) = c \cdot n(n+1)/2 = \Theta(n^2)$

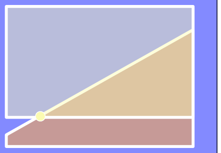
$$T(n) = \sum_{i=3}^{n} O(i) = O(n^2)$$



Complexity Analysis

- $T(n) = \sum_{i=3}^{n} c \cdot i = c(3+4+5+\ldots n) = c \cdot n(n+1)/2 = \Theta(n^2)$

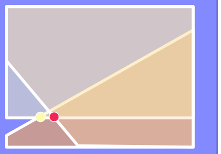
$$T(n) = \sum_{i=3}^{n} O(i) = O(n^2)$$



Complexity Analysis

- $T(n) = \sum_{i=3}^{n} c \cdot i = c(3+4+5+\ldots n) = c \cdot n(n+1)/2 = \Theta(n^2)$

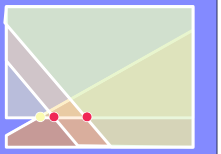
$$T(n) = \sum_{i=3}^{n} O(i) = O(n^2)$$



Complexity Analysis

- $T(n) = \sum_{i=3}^{n} c \cdot i = c(3+4+5+\ldots n) = c \cdot n(n+1)/2 = \Theta(n^2)$

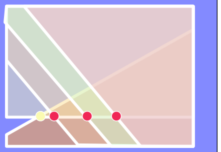
$$T(n) = \sum_{i=3}^{n} O(i) = O(n^2)$$



Complexity Analysis

- $T(n) = \sum_{i=3}^{n} c \cdot i = c(3+4+5+\ldots n) = c \cdot n(n+1)/2 = \Theta(n^2)$

$$T(n) = \sum_{i=3}^{n} O(i) = O(n^2)$$

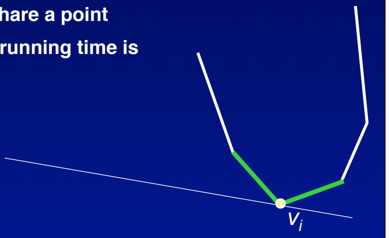


Incremental Algorithm – O(*n*) Randomized Version

- Exactly like the deterministic version, only the order of the lines is random.
- **Theorem:** The expected runtime of the random incremental algorithm (over all *n*! permutations of the input constraints) is O(*n*).

General Position Assumption

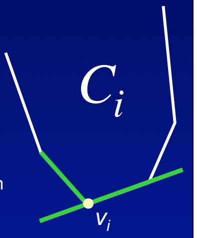
- **No three lines ℓ_i, ℓ_j, ℓ_k share a point**
- **If this is not the case, the running time is actually smaller.**



Probability Analysis

Backward analysis

- **Question**: When given a solution after *i* half-planes, what is the probability that the *last* half-plane affected the solution ?
- That is, $v_i \neq v_{i-1}$
- Remember also that in this case, v_i is on ℓ_i



Mental Trick: Assume that we decided who are the first i half-plane to be inserted. Lets look at C_i. It does not depend on the order of insertions.

What is the probability that the last inserted half-line is one of the two green ones ?

Backward analysis

- **Answer to the question**: Exactly, because a change can occur only if the last halfplane inserted is one of the two halfplanes thru *v_i*.

(note that v_i depends on the *i* half-planes, but not on their order)

Complexity Analysis

$$Pr(v_i \neq v_{i-1}) = \frac{2}{i}$$

Using the 1DLP, finding vi in this case takes i times (linear)

So the expected work at the i step is

$$1 \cdot Pr(v_i = v_{i-1}) + i \cdot Pr(v_i \neq v_{i-1}) = 1 * \frac{i-2}{i} + i \cdot \frac{2}{i} = 3$$

Lets do it again in a formal way

- We will use random variables. A random variable in our context will be a boolean value (flag) that is either true or false.
- Lemma (from probability): For any two constants A,B, and any two boolean random vars x,y, we define the expected value **E(xA+yB)** as
- $A \cdot Pr(x = 1 \text{ and } y = 0) + B \cdot Pr(x = 0 \text{ and } y = 1) + (A + B) \cdot (Pr(x = 1 \text{ and } y = 1)$
- However, we could greatly simplify it using the identity
- E(xA+yB)= $A \cdot Pr(x = 1) + B \cdot Prob(y = 1)$
- Note - we don’t care if x,y are depending in each other.
- In our case, lets define a set of boolean values
- $x_i = 1$ iff $v_i \neq v_{i-1}$.   Otherwise $x_i = 0$
- The running time of the algorithm is $\sum_{i=3}^{n} i \cdot x_i$

Lets do it again in a formal way

- We will use random variables. A random variable in our context will be a boolean value (flag) that is either true or false.
- Lemma (from probability): For any two constants A,B, and any two boolean random vars x,y, we define the expected value **E(xA+yB)** as
- $A \cdot Pr(x=1 \text{ and } y=0) + B \cdot Pr(x=0 \text{ and } y=1) + (A+B) \cdot (Pr(x=1 \text{ and } y=1)$
- However, we could greatly simplify it using the identity
- E(xA+yB)= $A \cdot Pr(x=1) + B \cdot Prob(y=1)$
- Note - we don't care if x,y are depending in each other.
- In our case, lets define a set of boolean values
- $x_i = 1$ iff $v_i \neq v_{i-1}$. Otherwise $x_i = 0$

Lets do it again in a formal way

The running time of the algorithm is $\sum_{i=3}^{n} i \cdot x_i$

The expected running time is $\mathbb{E}\{\sum_{i=3}^{n} i \cdot x_i\}$

Which (by applying the same rule multiple times)

$$\mathbb{E}\left\{\sum_{i=3}^{n} i \cdot x_i\right\} = \sum_{i=3}^{n} i \cdot Pr(v_i \neq v_{i-1}) = \sum_{i=3}^{n} 3 = 3n$$

LP in 3D

- Now the input is a collection of **half-spaces** $\{h_{1} \ldots h_n\}$.

Now l_i is the plane bounding h_i. (notations are analogous to the 2D case).

We will define v_3 as the intersection of the **planes** l_1, l_2 and l_3.

We insert the other halfspaces $\{h_4 \ldots h_n\}$ at a random order, and update v_i according to the following Theorem:

- Theorem:
  1. if $v_{i-1} \in h_i$, then $v_i = v_{i-1}$. // O(1) check, nothing to do
  2. if $v_{i-1} \notin h_i$, then the solution (if exists) is on l_i.
     run v_i = 2DLP($h_1 \cap l_i, h_2 \cap l_i, h_3 \cap l_i, \ldots, h_{i-1} \cap l_i$).
     Terminates if there is no solution (that is, $C_i = \emptyset$)

LP in 3D and higher dimension

In 3D, the worst case running time is ***Θ(n³) (prove).***

However, the expected running time is O(n). In general, the running time in d-dimension is O(d! n). That is, linear in any fixed (and small) dimension.