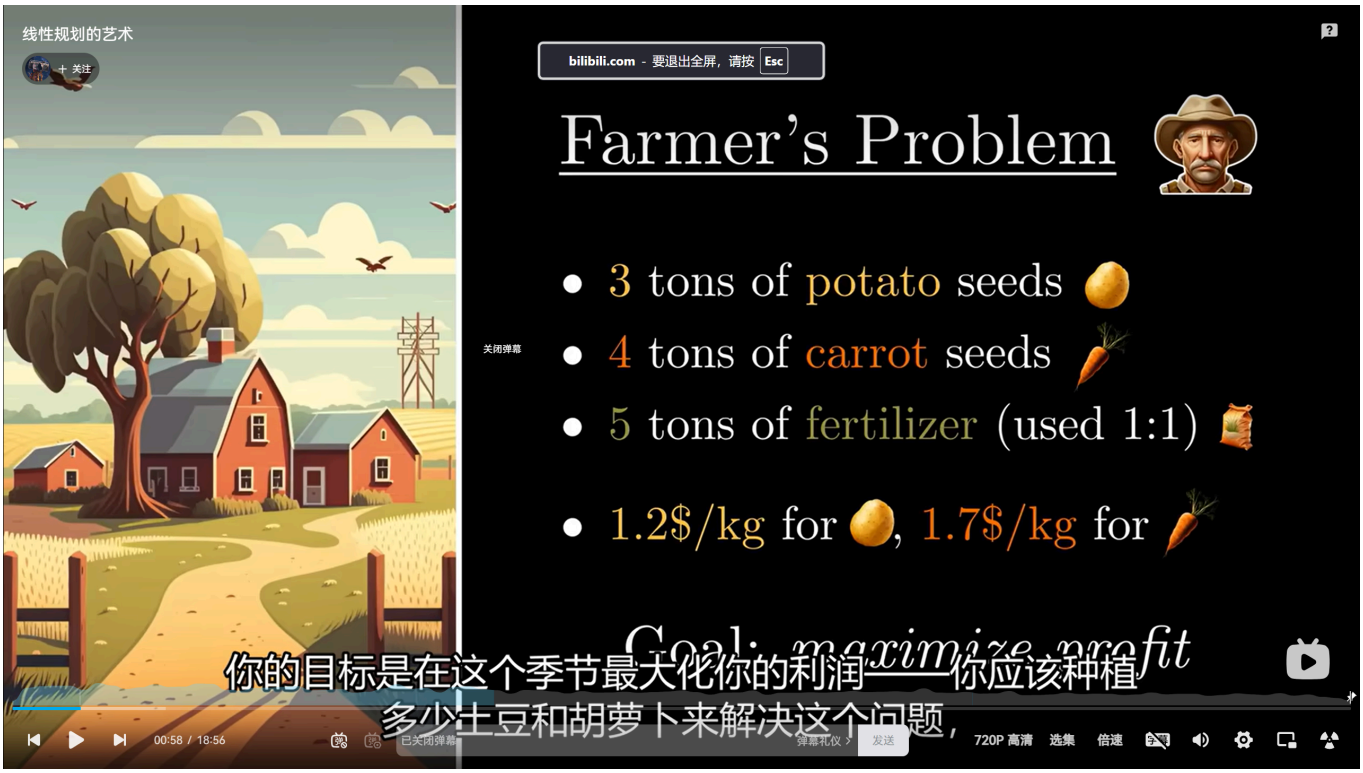
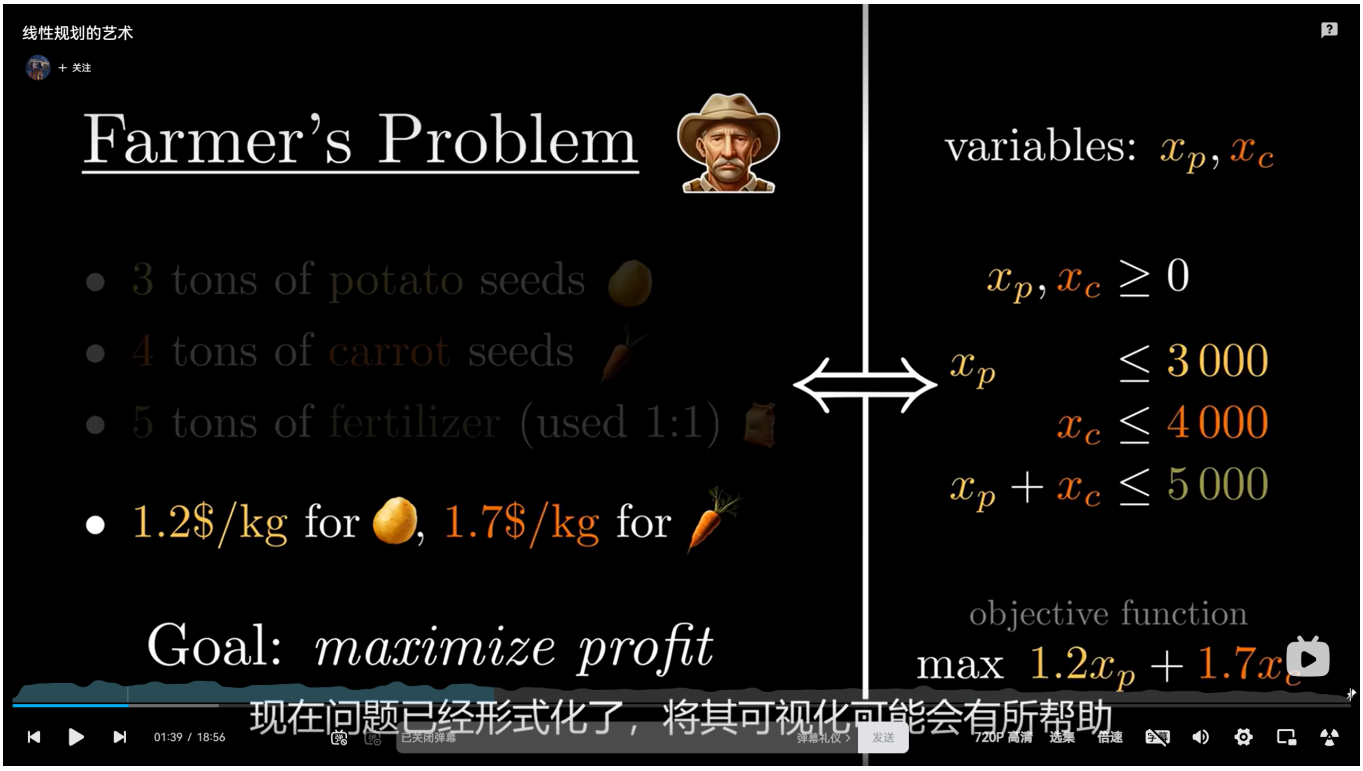
Linear Programming

introduce

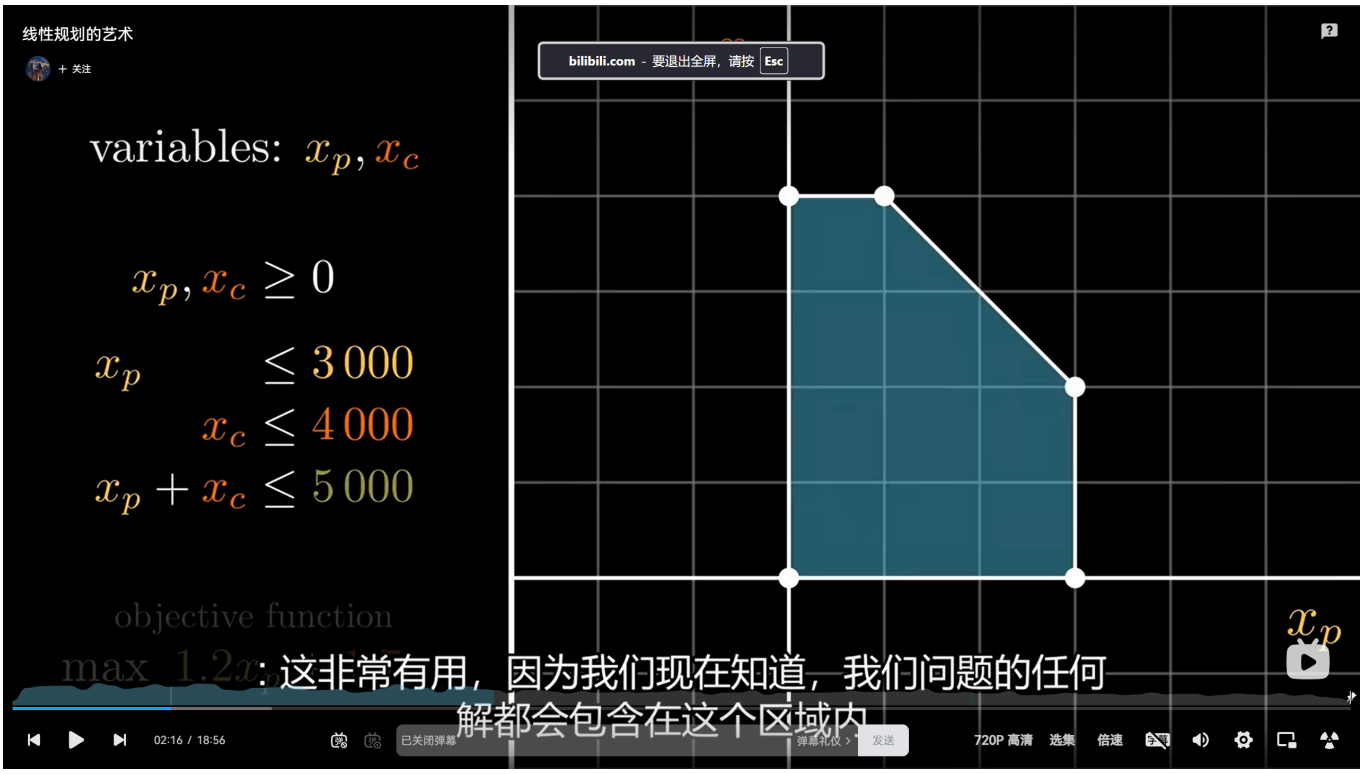


So, how many potatos and carrots should you plant as you want to maximize your interest?

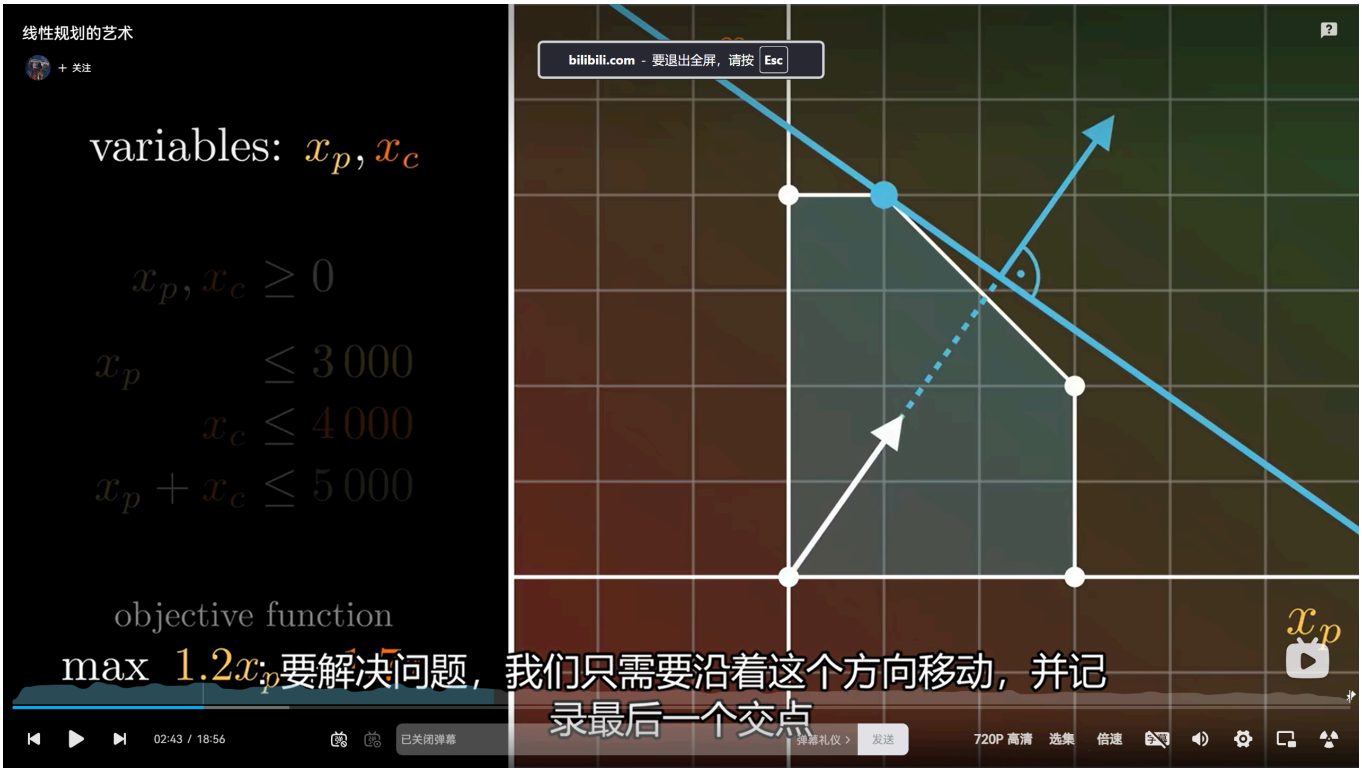
Now, we abstract it:



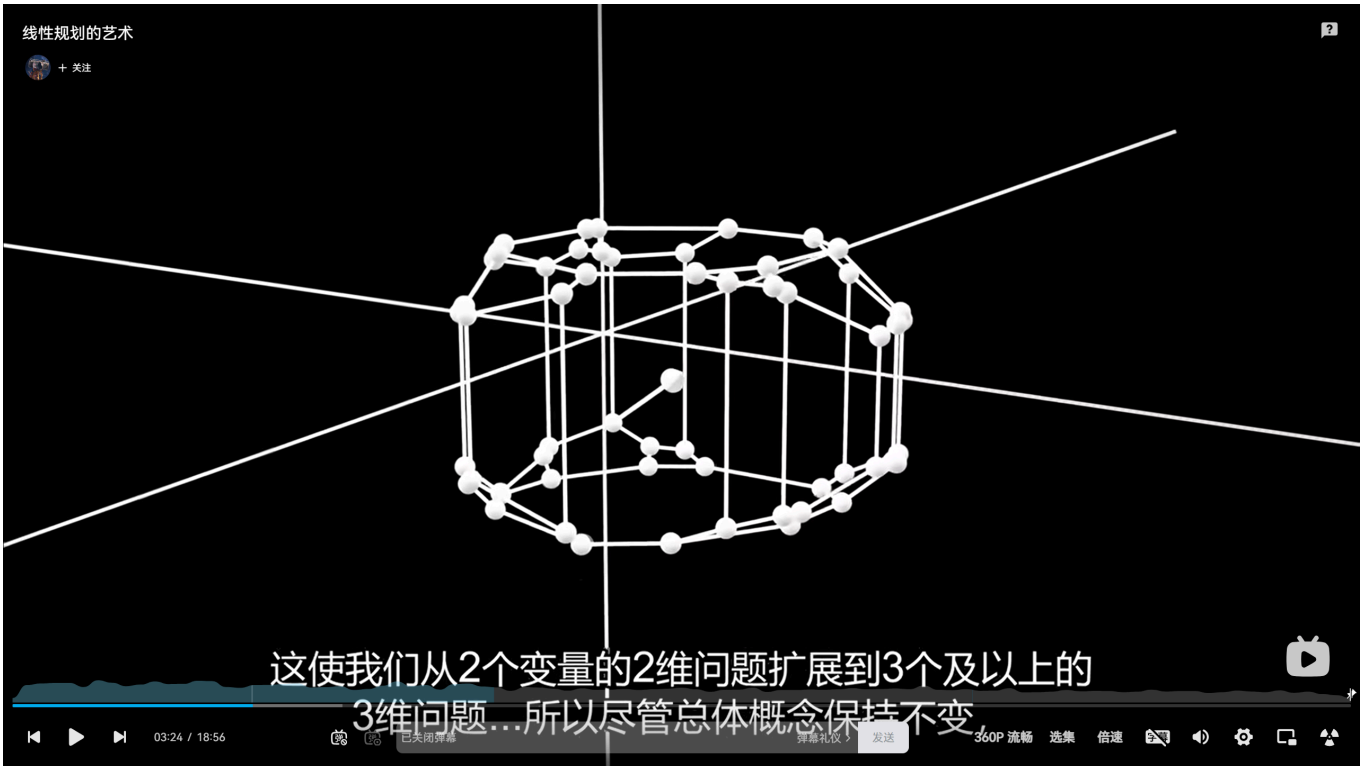
then let X_axis be Xp and let Y_axis be Xc.



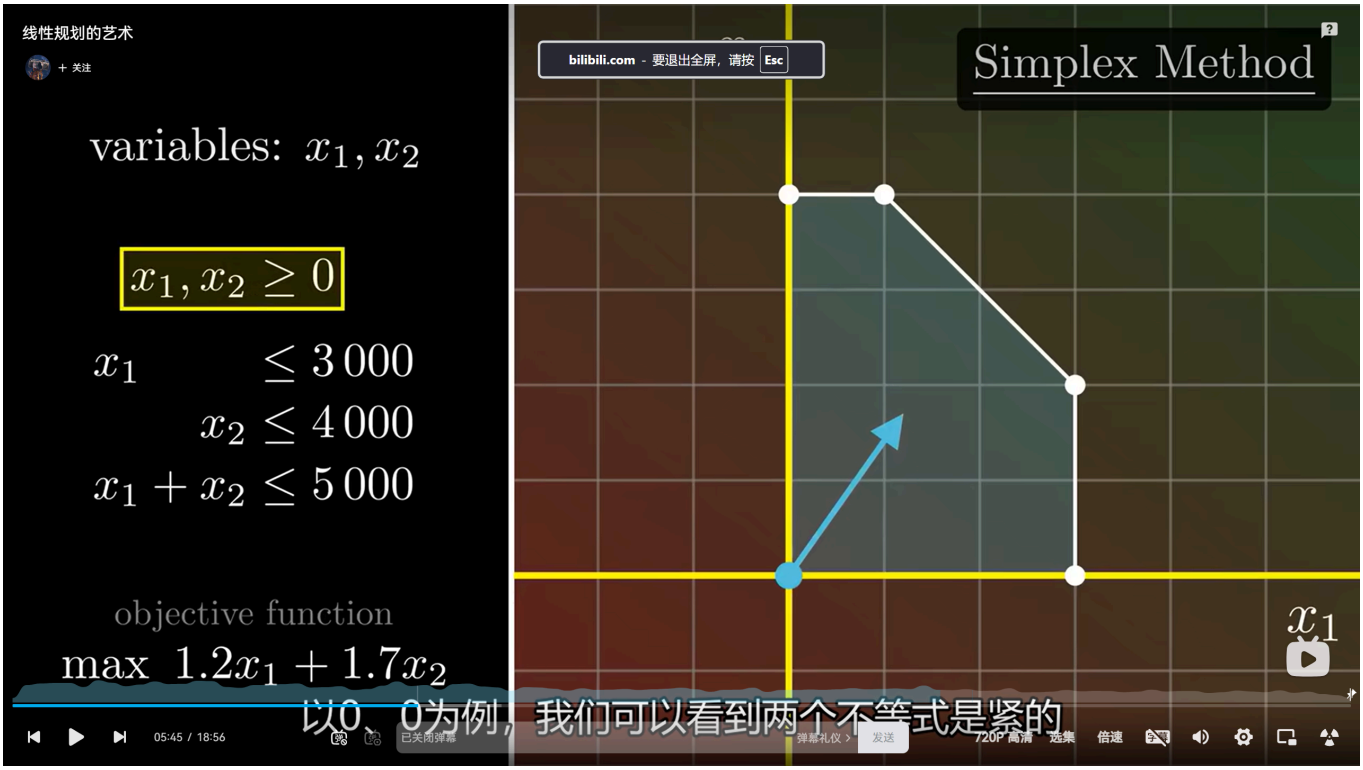
then move perpendicularly to 1.2Xp+1.7Xc.



But the problem will become difficult when the equations become more and more and when the independent variables become more and more. Just like this:



So let abstract it:



①: $x_1, x_2 \geq 0$

$x_1 \leq 3000$

$x_2 \leq 4000$

$x_1 + x_2 \leq 5000$

$\Rightarrow s_1 = 3000 - x_1$

$s_2 = 4000 - x_2$

$s_3 = 5000 - x_1 - x_2$

We want $1.2x_1 + 1.7x_2$

because

choose the biggest coefficient one to "loose"

$\Rightarrow \begin{cases} s_2 = 4000 - x_2 \\ s_3 = 5000 - x_1 - x_2 \end{cases}$ when one should we loose?

$\frac{4000}{-1} > \frac{5000}{-1}$, so loose $s_2 = 4000 - x_2$

$\Rightarrow$ then $x_2 = 4000 - s_2$ $\quad s_3 = 5000 - 4000 - x_1 + s_2 = 1000 - x_1 + s_2$

$1.2x_1 + 1.7 \cdot (4000 - s_2)$

$-1.7s_2 + 1.2x_1 + 6800$

$s_3 = 5000 - x_1 - x_2$

We want $1.2x_1 + 1.7x_2$

because

choose the biggest coefficient one to "loose"

$\Rightarrow \begin{cases} s_2 = 4000 - x_2 \\ s_3 = 5000 - x_1 - x_2 \end{cases}$ when one should we loose?

$\frac{4000}{-1} > \frac{5000}{-1}$, so loose $s_2 = 4000 - x_2$

$\Rightarrow$ then $x_2 = 4000 - s_2$ $\quad s_3 = 5000 - 4000 - x_1 + s_2 = 1000 - x_1 + s_2$

$1.2x_1 + 1.7 \cdot (4000 - s_2)$

$-1.7s_2 + 1.2x_1 + 6800$

then choose x_1 (as $1.2 > 1.7$)

$\Rightarrow \begin{cases} s_1 = 3000 - x_1 \\ s_3 = 1000 - x_1 + s_2 \end{cases}$ $\quad \frac{3000}{1} < \frac{1000}{-1}$ $\Rightarrow$ loose s_3

$x_1 = 1000 - s_3 + s_2 \Rightarrow -1.7s_2 + 1.2(1000 - s_3 + s_2) + 6800$

$= -0.5s_2 - 1.2s_3 + 8000$ $\Rightarrow$ 8000 is the answer

We got it !

Let see: We can transform the problem to this one:

let see:

$$y_1 \cdot x_1 \le 3000\, y_1$$

$$y_2 \cdot x_2 \le 4000\, y_2$$

$$y_3 (x_1 + x_2) \le 5000\, y_3$$

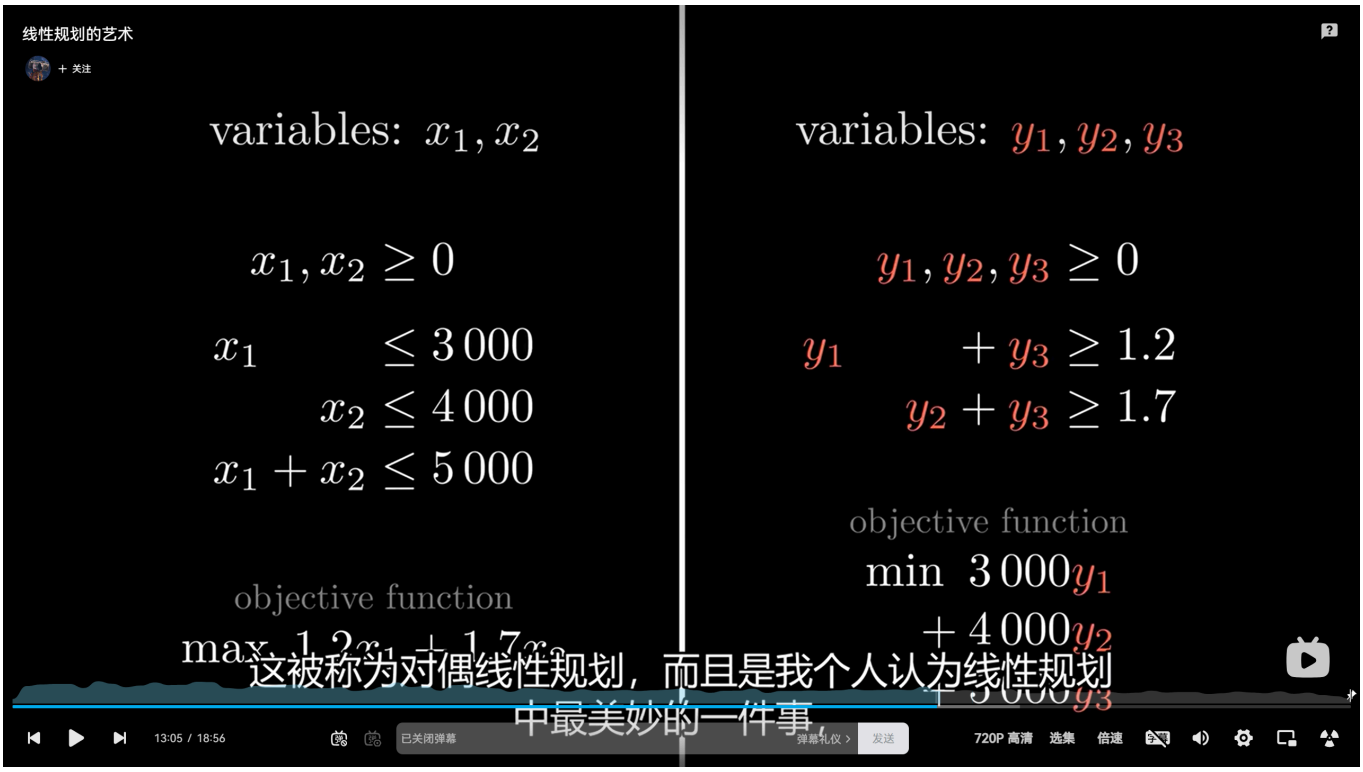
$\Rightarrow$ $y_1, y_2, y_3 \ge 0$

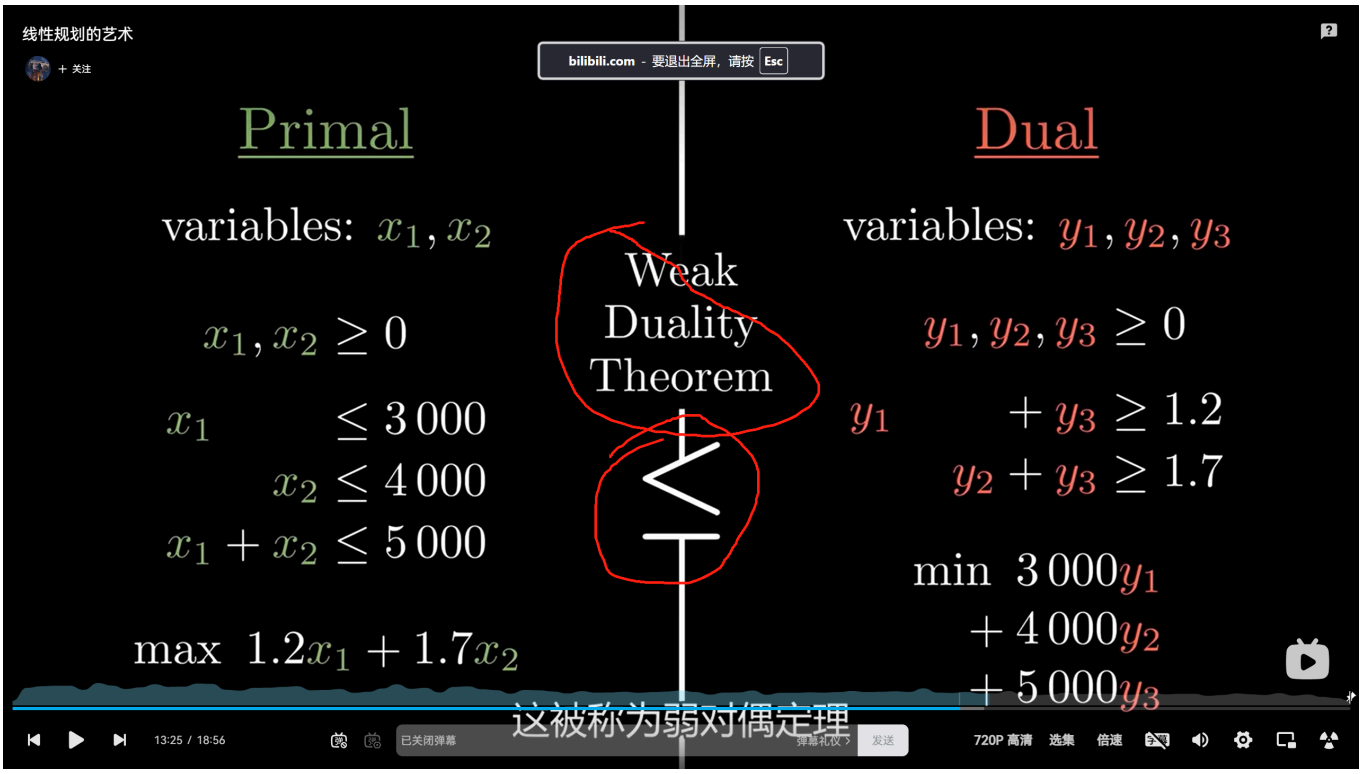
$$y_1 + y_3 \ge 1.2$$

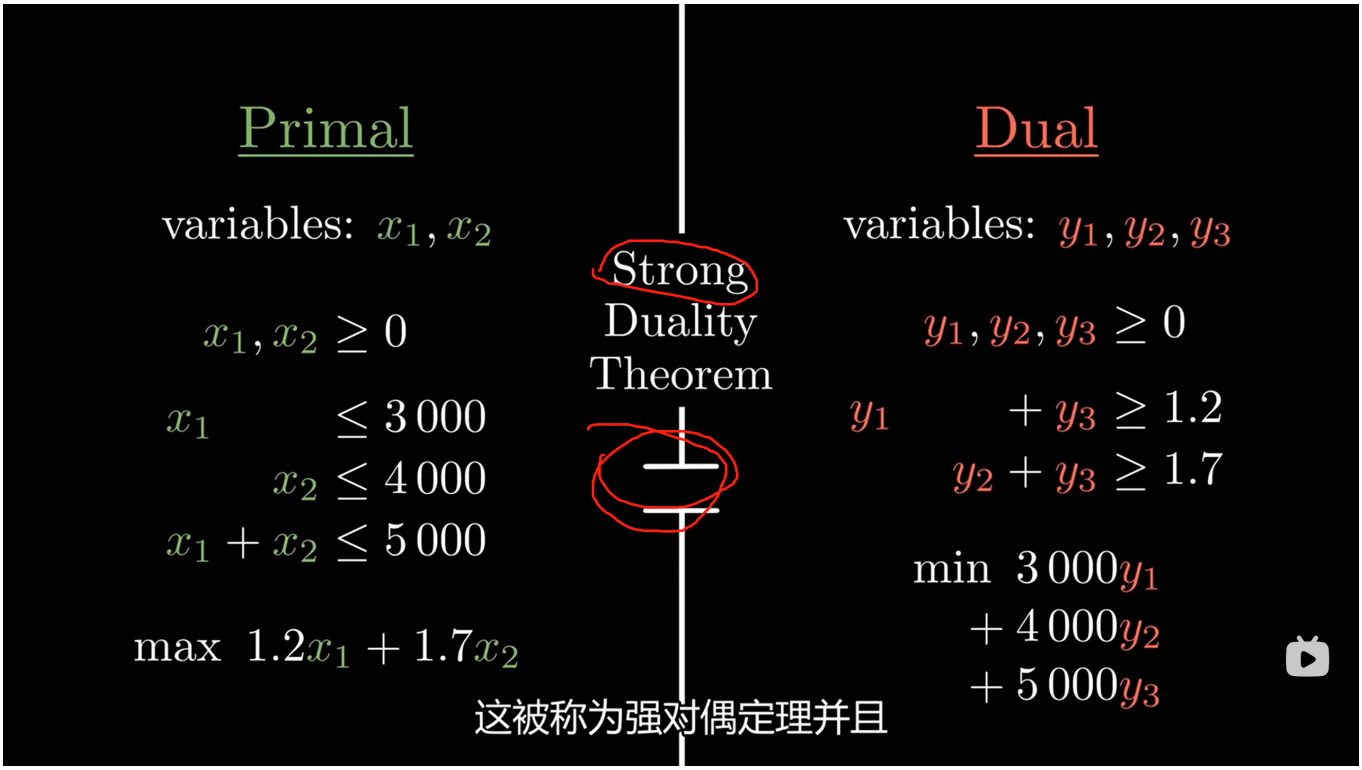
$$y_2 + y_3 \ge 1.7$$

$$\min 3000\, y_1 + 4000\, y_2 + 5000\, y_3$$

They are "symmetry".


线性规划的艺术
+ 关注
variables: x_1, x_2
$x_1, x_2 \geq 0$
$x_1 \leq 3\,000$
$x_2 \leq 4\,000$
$x_1 + x_2 \leq 5\,000$
objective function
$\max\ 1.2x_1 + 1.7x_2$
variables: y_1, y_2, y_3
$y_1, y_2, y_3 \geq 0$
$y_1 + y_3 \geq 1.2$
$y_2 + y_3 \geq 1.7$
objective function
$\min\ 3\,000y_1$
$+ 4\,000y_2$
$+ 5\,000y_3$
这被称为对偶线性规划，而且是我个人认为线性规划
中最美妙的一件事，
13:05 / 18:56
已关闭弹幕
弹幕礼仪 >
发送
720P 高清
选集
倍速





maximum flow and min-cut

EdmondsKarp:

```
#include<iostream>
#include<vector>
#include<queue>
```

```
using namespace std;

bool bfs(vector<vector<int>>&residualGraph,int source,int
sink,vector<int>&parent)
{
    int V = residualGraph.size();
    vector<bool> isvisited(V, false);
    queue<int> q;
    isvisited[source] = true;
    parent[source] = -1;
    q.push(source);
    while(!q.empty())
    {
        int u = q.front();
        q.pop();
        for (int v = 0; v < V; v++)
        {
            if(!isvisited[v]&&residualGraph[u][v]>0)
            {
                if(v==sink)
                {
                    parent[v] = u;
                    return true;
                }
                else
                {
                    isvisited[v] = true;
                    parent[v] = u;
                    q.push(v);
                }
            }
        }
    }
    return false;
}

int EdmondsKarp(vector<vector<int>>&graph,int source,int sink)
{
    vector<vector<int>> residualGraph = graph;
    int V = residualGraph.size();
```

```
    vector<int> parent(V);
    int maxFlow = 0;

    while(bfs(residualGraph,source,sink,parent))
    {
        int pathFlow = INT_MAX;
        for (int v = sink; v != source;v=parent[v])
        {
            int u = parent[v];
            pathFlow = min(pathFlow, residualGraph[u][v]);
        }
        maxFlow += pathFlow;
        for (int v = sink; v != source;v=parent[v])
        {
            int u = parent[v];
            residualGraph[u][v] -= pathFlow;
            residualGraph[v][u] += pathFlow;
        }
    }
    return maxFlow;
}

void findReachableVertices(vector<vector<int>>&residualGraph,int
source,vector<bool>&isvisited)
{
    int V = residualGraph.size();
    queue<int> q;
    q.push(source);
    isvisited[source] = true;
    while(!q.empty())
    {
        int u = q.front();
        q.pop();

        for (int i = 0; i < V;i++)
        {
            if(!isvisited[i]&&residualGraph[u][i]>0)
            {
                isvisited[i] = true;
                q.push(i);
```

```
            }
        }
    }
}

void findMinCut(vector<vector<int>>& graph,int source,int sink)
{
    int V = graph.size();
    vector<vector<int>> residualGraph = graph;
    vector<int> parent(V);

    int maxflow = EdmondsKarp(graph, source, sink);

    vector<bool> isvisited(V, false);
    findReachableVertices(residualGraph, source, isvisited);

    cout << "The edges in the minimum cut are:" << endl;
    for (int u = 0; u < V;u++)
    {
        cout << 111 << endl;
        for (int v = 0; v < V;v++)
        {
            cout << 22 << endl;
            if(isvisited[u]&&!isvisited[v]&&graph[u][v]>0)
            {
                cout << 33 << endl;
                cout << u << " - " << v << endl;
            }
        }
    }
}

int main()
{
    vector<vector<int>> graph =
    {
        {0, 16, 13, 0, 0, 0},
        {0, 0, 10, 12, 0, 0},
        {0, 4, 0, 0, 14, 0},
        {0, 0, 9, 0, 0, 20},
```

```
        {0, 0, 0, 7, 0, 4},
        {0, 0, 0, 0, 0, 0}
    };

    int source = 0;
    int sink = 5;

    cout << "The maximum possible flow is " << EdmondsKarp(graph, source,
sink) << endl;
    findMinCut(graph, source, sink);
    return 0;
}
```

Dinic:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

class Dinic
{
public:
    Dinic(int n)
    {
        this->n = n;
        this->level.resize(n);
        this->adj.resize(n);
    }
    void addEdge(int u, int v, int capacity)
    {
        adj[u].push_back(edges.size());
        edges.push_back({u, v, capacity, 0});
        adj[v].push_back(edges.size());
        edges.push_back({v, u, 0, 0});
    }

    int maxflow(int source, int sink)
    {
        int flow = 0;
```

```
        while (bfs(source, sink))
        {
            while (int pushed = dfs(source, sink, INT_MAX))
            {
                flow += pushed;
            }
        }
        return flow;
    }

private:
    int n; // number of vectices
    int source;
    int sink;
    struct Edge
    {
        int u, v, capacity, flow;
    };
    vector<Edge> edges;
    vector<int> level;
    vector<vector<int>> adj;

    bool bfs(int source, int sink)
    {
        fill(level.begin(), level.end(), -1);
        queue<int> q;
        q.push(source);
        level[source] = 0;
        while (!q.empty())
        {
            int u = q.front();
            q.pop();
            for (int id : adj[u])
            {
                Edge &e = edges[id];
                if (e.capacity > e.flow && level[e.v] == -1)
                {
                    level[e.v] = level[u] + 1;
                    q.push(e.v);
                }
```

```
            }
        }
        return level[sink] != -1;
    }

    bool dfs(int u, int sink, int pushed)
    {
        if (pushed <= 0)
        {
            return 0;
        }
        if (u == sink)
        {
            return pushed;
        }
        for (int i = 0; i < adj[u].size(); i++)
        {
            int id = adj[u][i];
            Edge &e = edges[id];
            if (level[u] + 1 != level[e.v] || e.capacity <= e.flow)
            {
                continue;
            }
            int tr = dfs(e.v, sink, min(pushed, e.capacity - e.flow));
            if (tr <= 0)
            {
                continue;
            }

            e.flow += tr;
            edges[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }
};

int main()
{
    int n = 6; // Number of vertices
```

```
    Dinic dinic(n);

    // Add edges
    dinic.addEdge(0, 1, 16);
    dinic.addEdge(0, 2, 13);
    dinic.addEdge(1, 2, 10);
    dinic.addEdge(1, 3, 12);
    dinic.addEdge(2, 1, 4);
    dinic.addEdge(2, 4, 14);
    dinic.addEdge(3, 2, 9);
    dinic.addEdge(3, 5, 20);
    dinic.addEdge(4, 3, 7);
    dinic.addEdge(4, 5, 4);

    int source = 0;
    int sink = 5;

    cout << "The maximum possible flow is " << dinic.maxflow(source, sink) <<
endl;

    return 0;
}
```