Listing 1: **Construct.m**

```
function Construct(t)
%  This function is used to construct the price-subsidy function.
%  t could be set to [7, 6.5,6.5,4,4,3.5,2.5,2, 1.5,1];
interval = Pretreatment(t);
x = []; % Restore the value of price at the breakpoints.
y = []; % Restore the value of subsidy at the breakpoints.

for i = 1: length(interval)-1
  aa = interval(i, 1: 2);
  [list_x, list_y] = IPCtest(t, [aa(1), aa(2)- 0.5]);
  x = [x, list_x];
  y = [y, list_y];
end

x = [x, 0, interval(1, 2)];
y = [y, 0, 0];
point_xy = [x; y];
point_xy = sortrows(point_xy', 1)';

plot(point_xy(1, :),point_xy(2, :));
saveas(gcf, 'IPC.pdf');

end
```

Listing 2: **Pretreatment.m**

```
function [I, price] = Pretreatment(t)
% This function is used to obtain the interval of price given the
    initial processing time of each player.
% t is given in a descending order.
% Return a list of interval.
% The first two elements are the interval of price, the third one is
    the corresponding number of machines.
t1 = fliplr(t); % Reverse the list
N = length(t);

cV = zeros(1, ceil(N/2)+ 1);
```

```
% This loop is used to obtain the cV(i)
for k = 1: ceil(N/2)+ 1
    s = floor(N/k);
    r = rem(N, k);
    a = (s+1): -1: 1;
    repeat = k;
    tmp = repmat(a, repeat, 1);
    b = reshape(tmp, 1, length(a)* repeat);
    b(1: k-r) = [];
    cV(k) = dot(b, t1);
end

price = diff(fliplr(cV));  % [p_n/2, p_n/2-1,...,p_2]
p_1 = dot(0: N-1, t);

price(end+1) = p_1;
price = [0, price];
I = zeros(ceil(N/2)+ 1, 3);

for i = 1: ceil(N/2)+ 1
  I(i, :) = [price(ceil(N/2)- i+ 2), price(ceil(N/2)- i+ 3), i];
end

end
```

Listing 3: **Pm.m**

```
function m = Pm(P, t)
% Give the price then return the corresponding number of machines.
[interval, price] = Pretreatment(t);
a = 1;
b = length(price);
c = floor((a + b)/2);
flag = 1;

if (P < 0) | (P > price(end))
  disp('Wrong in Pm');
  return
```

```
end

if abs(P- price(end)) < 1e-5
  m = 1;
  return
end

while flag
  if price(c) <= P
    if P < price(c + 1)
      m = b- interval(c, 3);
      flag = 0;

    else
      c = floor((c + b)/2);
    end

  else
    c = floor((a + c)/2);
  end
end

end
```

Listing 4: **TotalCost.m**

```
function cV = TotalCost(t, P)
% This function returns total cost when using m machines and price is P.
t1 = fliplr(t); % reverse the list
N = length(t);
k = Pm(P, t);

s = floor(N/k);
r = rem(N, k);
a = (s+ 1): -1: 1;
repeat = k;
tmp = repmat(a, repeat, 1);
```

```
b = reshape(tmp, 1, length(a)* repeat);
b(1: k-r) = [];
cV = dot(b, t1) + k * P;

end
```

Listing 5: **IPC.m**

```
function [Pstar, omega] = IPCtest(t, Pbig)
% Pretreatment: Give all the sub-intervals [P_m+1,P_m]
% Return intersection set and the corresponding subsidy.
% Then we just need connect these points.
% t = [7.5,6,5.5,4,3,1.5,1.5,1.5];
% Pbig is the interval of price and will gradually decrease
v = length(t);
Pstar = Pbig; % The set of breakpoints
omega = zeros(1, 2);
count = 0;

while ~isempty(Pbig)
  [a1, b1, c1] =  CP(v, t, Pbig(1, 1));   % omega  K_l  K_r
  [a2, b2, c2] =  CP(v, t, Pbig(1, 2));

  if count < 0.5
    omega(1) = a1;
    omega(2) = a2;
  end

  count = count + 1;
  slope =  (a2- a1)/(Pbig(1, 2)- Pbig(1, 1)); % The value is Negative.
  % (z_k-1 select K_r) / (z_k  select K_l)
  if (round(b2, 5) == round(slope, 5))||(abs(c1-slope)< 1e-5)
    Pbig(1, :) = [];
  else
    % How to calculate the intersection point
    zinter = (c1*Pbig(1, 1) - Pbig(1, 2)*b2 + a2 - a1)/(c1- b2);
    omega1 = (zinter - Pbig(1, 2))* b2 + a2;
    [a, b, c] = CP(v, t, zinter);
```

```

    if abs(omega1-a) < 1e-5 % Two subsidy equal, this is the
        breakpoint,
    % then delete this interval, store the breakpoint
      omega = [omega, omega1];
      Pstar = [Pstar, zinter];
      Pbig(1, :) = [];

    else
      Pbig(end+1, :) = [Pbig(1,1), zinter];
      % Notice that there is already add a new row
      Pbig(end+1, :) = [zinter, Pbig(1,2)];
      Pbig(1, :) = [];
    end
  end
end

end
```

Listing 6: **CP.m**

```
function [omega, K_l, K_r] = CP(v, t, P)
% Set the initial coalition s = {{1},{2},{3}...{v}} or |s|=v-1
% beta is the optimal solution.
% opt_s is a optimal vector solution s.
% t is the time cost for every player.
% v is number of players.
% P is the price.
% m is the number of optimal using machines.
% return subsidy and min / max slope

ini_s = 1 - eye(v);
cV = TotalCost(t, P);  % c_V = m*P +sum_t;

flag = true;
count = 0;

while flag
```

```
  [beta, maxr] = LP2(ini_s, v, t, P);
  [delta, opt_s] = DP(v, t, beta,P);

  if delta < -0.001
    ini_s = [ini_s; opt_s];

  else
    omega = cV - maxr;
    % Here use Coalition obtain all maximum unsatisfied coalitions
    unsatisfied = Coalition(ini_s, v, t, P);
    [K_l, K_r] = LP1(unsatisfied, t, P);
    flag = false;
  end

  count = count + 1;
  if count > 100
      disp('There is something wrong in CP')
      break
  end
end

end
```

Listing 7: **DP.m**

```
function [res, s] = DP1(v, t, beta, P)
% v is number of players
% t and beta are vectors from bottom1 to topN
% For example, P = 9.5
% t = [5 4 3 2];
% beta = [14.5 8 12.5 4];
% Notice could not exist V (u \neq v) here.
% s is the optimal solution.
P = ones(v, v+1);
P(1, 1) = P;  % P(1, 0) in DP algorithm
P(1, 2) = P + t(1) - beta(1);  % P(1, 1) in DP algorithm

ss = cell(v, v+ 1);  % Used to store the corresponding player vector.
```

```
ss(1, 1) = {Peros(1, 1)};
ss(1, 2) = {ones(1, 1)};

for i = 2: v
  % s = ss(:,);  %  every loop only records this column
  P(i, 1) = P(i-1, 1);
  ss(i, 1) = {Peros(1, i)};

  P(i, i+1) = P(i-1, i) + i* t(i) - beta(i);
  ss(i, i+1) = {ones(1, i)};

  for j = 1: i-1
    if P(i-1, j+1) > (P(i-1, j)+ j* t(i)- beta(i))
      P(i, j+1) = P(i-1, j)+ j* t(i)- beta(i); % notice that j is not
          the ordinal number
      ss(i, j+1) = {[ss{i-1, j}, 1]};

    else
      P(i, j+1) = P(i-1, j+1);  %
      ss(i, j+1) = {[ss{i-1, j+1}, 0]};
    end
  end

end

[res, ind] = min(P(v, 2: v)); % Record the P(v,u) u \in (1:v-1)
s = ss{v, 1+ind};

end
```

Listing 8: **Coalition.m**

```
function unsatisfied = Coalition(s, v, t, z)
% This function is used to obtain all the unsatified coalitions.
% Notice that s is a restricted-coalition matrix.(0-1)
% Which is obtained from CP method.
% t is a column vector arranged from large to small.
% Return all the unsatified coalitions.
```

```
s1 = length(s(:, 1));  % The number of constraints
c_s = zeros(s1, 1);

for i = 1: s1
    tot = sum(s(i, :) == 1);
    inde = s(i,:) == 1;  % May not use the function 'find'
    c_s(i) = dot(1: tot, t(inde)) + z;
end

f = ones(v, 1);
b = c_s;
% optimset('Display','off');
[x, fval, exitflag, output, lambda] = linprog(-f, s, b);
inde = lambda.ineqlin ~= 0;
normal_order = (1: s1)';
normal_order = normal_order(inde);
unsatisfied = s(normal_order, :);

end
```

Listing 9: **LP1.m**

```
function [minr, maxr] = LP9(s, t, P)
% This function is used to give the min and max slope value.
% Notice that s is a matrix. (0-1)
% For example, v = 4
% s=[0 1 1 0;
%    0 1 0 1;
%    0 0 1 1;
%    1 1 1 0;
%    1 1 0 1;
%    1 0 1 1;]
v = length(t);
m_v = Pm(P, t);   % Notice that the result has to add m_v.
s1 = length(s(:, 1));
f = ones(s1, 1);
b = ones(v, 1);
lb = zeros(s1, 1);
```

```
ub = ones(s1, 1);

[x, fval1] = linprog(-f, [], [], s', b, lb, ub);
minr = m_v+ fval1;
[x, fval2] = linprog(f, [], [], s', b, lb, ub);
maxr = m_v- fval2;

end
```

Listing 10: **LP2.m**

```
function [x, maxr] = LP12(s, v, t, z)
% Notice that s is a restricted-coalition matrix.(0-1)
% Which is obtained from CP method.
% t is a column vector arranged from large to small.
% Return the solution vector x and value maxr.
s1 = length(s(:, 1));  % The number of constraints
c_s = zeros(s1, 1);

for i =1: s1
    tot = sum(s(i, :) == 1);
    inde = s(i, :) == 1;  % May not use the function 'find'
    c_s(i) = dot(1: tot, t(inde)) + z;
end

f = ones(v, 1);
b = c_s;
[x, fval1] = linprog(-f, s, b);
maxr = -fval1;

end
```

Listing 11: **Players.m**

```
function [an, index, Subsidy, difference] = Players(n, S0, an)
% At first, we should set up an array to storage the number we need
    later.
% n is the quantity of players, and m is the number of machine.
```

```
% And we suppose m>n.
ann = []; %  ann will be a matrix whose dimension is (2^n-1,n).
A = eye(n);
f = (-1)* ones(n, 1);
% ff2n: Two-level full factorial design
a1 = ff2n(n);
b1 = sum(a1, 2);
c1 = [a1, b1];
d1 = sortrows(c1, [n+1 1:n], 'descend');
e1 = sortrows(d1, n+1);
A = e1(:, 1: n);
A(1, :) = [];  % delete the first row

for i = 1 : n   % i is the number of players
%  In the first loop, find all combinations of k players and put it in
     an ascending order by default.
    bn = zeros(1, nchoosek(n, i)); % Store the final result.
    dn = zeros(1, i);
    cn = nchoosek(an, i);        % Store the temporary sort result.
    for j = 1: nchoosek(n, i)
        %  The second loop obtain the result in every coalition.
        for k = 1: i     % k in the number of machines.
            s = floor(i/k);  % Obtain the quotient.
            r = rem(i,k);
            a = (s+1): -1: 1;
            repeat = k;   % Repeat the number of machines.
            tmp = repmat(a, repeat, 1);
            b = reshape(tmp, 1, length(a)* repeat);
            b(r+1: k) = [];
            dn(k) = dot(b, cn(j, :)) + k* S0;
        end

        [minofdn index] = min(dn);  % Find the minimum.
        bn(j) = min(dn);
    end

    ann = [ann bn];
end

```

```
[x,y] = linprog(f,A,ann);
Subsidy = ann(end)+y;
Taxation = index * S0 ;
difference = Subsidy - Taxation;

end
```