Problem 1:
Answer: When the length n = 4, according to the "greedy" strategy, it is cut into steel bars of length 1 and 3 separately. So, p = 1 + 8 = 9. And the optimal solution is when it is cut into 2 steel bars of length 2. The optimal value is: p = 5 + 5 = 10> 9.

Problem 2:
Answer: Create a new array m [0 ... n] to record the number of cutting segments of the optimal solution for each length of steel bar. When the optimal solution for the steel bar of length i is completed, update m [i +1] = m [j] + 1, where the steel bar of length (i + 1) is cut into two large sections of length (i + 1-j) and j. And the steel bar of length j continues to be cut.

Algorithm is following:

```
Let r[0...n] and m[0...n] be new arrays
  r[0] = 0, m[0] = 0
  for i = 1 to n
    q = -∞
    for j = 1 to i
      if q < p[j] + r[i-j] - m[i-j]*c
        q = p[j] + r[i-j] - m[i-j]*c
        m[i] = m[i-j] + 1
    r[i] = q
  return r[n]
```

Problem 3:
Answer:
a)
Dynamic programming method pseudocode:

```
// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity (W)
Function Knapsack_dp(W , wt , val , n)
{//return the maximum value that could be put into knapsack
for j = 0 to W do:
 if w[i] < j then: m[n, j] := 0
 else: m[n, j] := v[n]
//build table k[][] from the bottom to the top
for i = n - 1 downto 1 do:
  for j from 0 to W do:
     if w[i] <= j then:
        m[i, j] := max(m[i + 1, j], m[i + 1, j - w[i]] + v[i])
        else:
            m[i, j] := m[i + 1, j]
            }
```

Recursive method pseudocode:

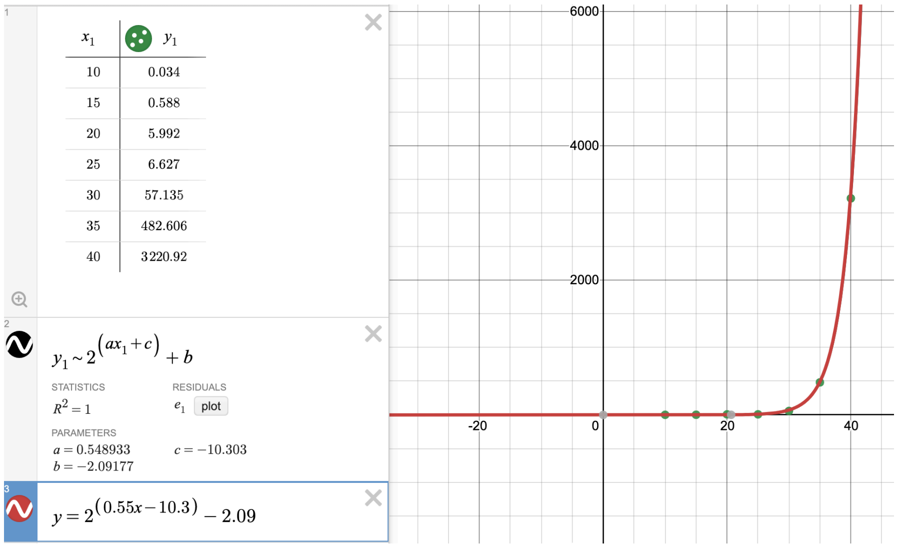
```
Function knapsackRecursive(W , wt , val , n){
//Base Case
  if n == 0 or W == 0 :
      return 0
  // If weight of the nth item is more than the Knapsack of capacity W,
  // this item should be ignored
   if (wt[n-1] > W):
      return knapSack(W , wt , val , n-1)

   // return the maximum of two cases: nth item included, not included
   else:
      return max(val[n-1] + knapSack(W-wt[n-1] , wt , val , n-1),
                 knapSack(W , wt , val , n-1))
 }
```

b) In teach

c) the data curve and the expressions are following.

N	recursive method		N	dynamic programming
10	0.034		10	0.02
15	0.588		15	0.036
20	5.992		20	0.047
25	6.627		25	0.071
30	57.135		30	0.082
35	482.606		35	0.109
40	3220.92		40	0.121

Recursive method:



x_1	y_1
10	0.034
15	0.588
20	5.992
25	6.627
30	57.135
35	482.606
40	3220.92

$y_1 \sim 2^{(ax_1+c)} + b$

STATISTICS $R^2 = 1$

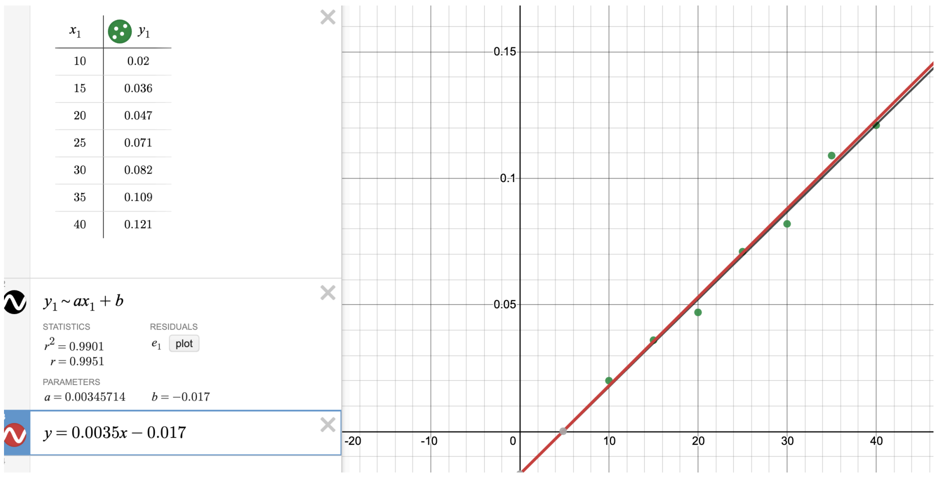
RESIDUALS e_1 plot

PARAMETERS

$a = 0.548933$ $c = -10.303$

$b = -2.09177$

$y = 2^{(0.55x-10.3)} - 2.09$

Dp method: 

x_1	y_1
10	0.02
15	0.036
20	0.047
25	0.071
30	0.082
35	0.109
40	0.121

$y_1 \sim ax_1 + b$

STATISTICS

$r^2 = 0.9901$

$r = 0.9951$

RESIDUALS

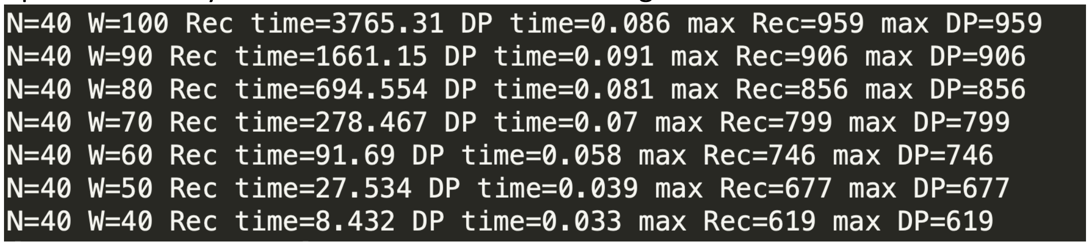
e_1 plot

PARAMETERS

$a = 0.00345714$ $b = -0.017$

$y = 0.0035x - 0.017$

d) My implementation of the program is C++. The val is generated randomly, the range is [1,100], and the weight is random generated from [1,30]. I found the time complexity of dynamic programming method executes linear, which is O(Kn). The time complexity of recursive method executes exponentially, which is O(2^n). I collected it with desmos. Additionally, with the W growing, the execute time of recursive will be exponential. The dp time is nearly linear. The test value is following:



```
N=40 W=100 Rec time=3765.31 DP time=0.086 max Rec=959 max DP=959
N=40 W=90 Rec time=1661.15 DP time=0.091 max Rec=906 max DP=906
N=40 W=80 Rec time=694.554 DP time=0.081 max Rec=856 max DP=856
N=40 W=70 Rec time=278.467 DP time=0.07 max Rec=799 max DP=799
N=40 W=60 Rec time=91.69 DP time=0.058 max Rec=746 max DP=746
N=40 W=50 Rec time=27.534 DP time=0.039 max Rec=677 max DP=677
N=40 W=40 Rec time=8.432 DP time=0.033 max Rec=619 max DP=619
```

Problem 4:

Answer:

a) The question could convert into 01- knapsack problem, which is optimal values. Total price is the limit W, weight is w, each item's price is val, n is the max weight each person can carry. It means that we could create a same logic dynamic programming algorithm to maximum the total price.

Pseudocode:

```
Function Knapsack_dp(W , w , val , n)
{//return the maximum value that could be put into knapsack
for j = 0 to W do:
 if w[i] < j then: m[n, j] := 0
 else: m[n, j] := v[n]
//build table k[][] from the bottom to the top
for i = n - 1 downto 1 do:
   for j from 0 to W do:
       if w[i] <= j then:
           m[i, j] := max(m[i + 1, j], m[i + 1, j - w[i]] + v[i])
else:
    m[i, j] := m[i + 1, j]
    }
```

b) In theoretical running time is O(N*$\sum_{i=1}^{F} M_i$)where N is the given item. The M will reach top when each person got their personal max value Mi, which means the capacity of bag is M1+M2+…+Mf, 1≤i≤F.
c) in teach