Homework #2

Reference Solutions

Contact TAs: `ada-ta@csie.ntu.edu.tw`

Problem 1 - Pusheen the Cat (Programming) (10 points)

Problem Description

Pie, a snack lover, notices that there are cupcakes appearing randomly in his house recently. After many efforts (including eating cupcakes), he finds out that these cupcakes are produced by his pet cat, Pusheen.

After some survey about Pusheen, Pie finds a book named "Assistant for Dark-magic Access", which explains how Pusheen produces cupcakes using dark magic (*not* from its digestive system). The book says that a level-k cat with dark magic will produce 2^k cupcakes every day, and Pusheen is currently a level-0 cat. If you feed Pusheen with ramen *after it produces cupcake(s)*, it will level up! That is, a level-k cat will become level-$(k+1)$ after consuming a bowl of ramen. You can also feed Pusheen with Big Mac and remain its level. Note that Pusheen can only eat one bowl of ramen or one Big Mac per day and it produces cupcake(s) *every day*.

After knowing this, Pie decides to get cupcakes from Pusheen! However, after considering the high calories of cupcakes, Pie decides to have *exactly* N cupcakes from Pusheen. Pie wants to find out the minimum day(s) needed to get N cupcakes starting from a level-0 Pusheen. Can you help him find the answer? As a reward, Pie will give you the score of this problem.

Input

The first line of the input file contains an integer T, indicating the number of testcase(s). Each testcase contains an integer N, indicating the desired number of cupcake(s).

- $1 \le T \le 200,000$
- $1 \le N \le 1,000,000,000$

Subtask 1 (40 %)

- $N \le 1,000$

Subtask 2 (60%)

- no other constraints.

Output

For each testcase please output a line with an integer indicating the minimum day(s) needed to reach N cupcakes.

Sample Input

```
4
1
13
127
4610
```

Sample Output

```
1
5
7
15
```

Although the problem said that a level-k pusheen can only produce 2^k cupcakes every day, you can assume that it can produces one of $1, 2, 3, \ldots, 2^k$ cup cakes. Because a level-k pusheen comes from level-0, if you want it produces 4 cup cakes, it is equivalent to that you make pusheen stay at level-2 for 1 day longer.

With this concept, we can find an algorithm: keep doubling the cup cakes until it exceeds N. If this happens after d days, let R be the rest part, i.e.,

$$R = N - \sum_{i=0}^{d} 2^i.$$

Then because $R \leq 2^d$, the minimum days needed is the number of 1's in R's binary representation.

This algorithm can find the optimal solution. Let a solution be a sequence of powers of 2. The optimal solution must reach the same maximum value, i.e., 2^d defined in previous paragraph. If it didn't, then there must exist a k such that 2^k appears more than 3 times in the sequence and you can replace 2 of them with a 2^{k+1} to find a better solution. If a sequence meet the same maximum, then for the rest part, binary representation is the quickest way to get that much cup cakes.

The code by problem setter is here: http://codepad.org/ZOaoBSxi

Problem 2 - ADA Farm (Programming) (15 points + Bonus 3 points)

Problem Description

Have you ever heard of the ADA farm? It is a great place to visit with your friends (if any)!

The ADA farm is well-known for the enormous number of horses, and you will agree with this after reading the "Input" section. There are N horses in total on the ADA farm, where each horse lives in a different position. However, you noticed that some horses look lonely. After observing them for a while, you know that the horses often visit their friends during their free time. If other horses are really far away, then visiting them becomes time-consuming and tiring so that the horse will be upset.

The way horses move is really unusual. If you consider the ADA farm as a 2-dimensional space, then a horse at coordinate (x, y) has 2 options to move:

1. The well-known way: the horse can spend 2 seconds and jump to one of $\{(x+2, y+1), (x+2, y-1), (x+1, y-2), (x-1, y-2), (x-2, y-1), (x-2, y+1), (x-1, y+2), (x+1, y+2)\}$, like the normal horses.
2. The unusual way: the horse can spend 1 second and walk to one of $\{(x+1, y), (x, y+1), (x-1, y), (x, y-1)\}$, just like a soldier! It's really unbelievable, isn't it?

We assign numbers for these horses from 1 to N with the i-th horse h_i living at a position (x_i, y_i). Then we can define a *loneliness value* L for each horse h:

$$L(h) = \sum_{i=1}^{n} d(h, h_i),$$

where $d(a, b)$ is the minimum time cost between the position of the horse a and the horse b. That is, the loneliness of a horse is the sum of the minimum time needed for it to move from its home to another horse's home.

In order to know more about the horses, can you find the loneliness value of each of N horses in the ADA farm?

Input

The first line of the input file contains an integer N, indicating the number of horses in the ADA farm.

For the next N lines, the i-th line contains 2 integers x_i, y_i, indicating the position of i-th horse in the ADA farm.

- $2 \le N \le 100,000$
- $0 \le x_i, y_i \le 1,000,000$

Subtask 1 (70 %)

- $N = 2$

Subtask 2 (30 %)

- $N \le 1000$

Subtask 3 (Bonus, 20%) (Very Difficult!)

- no other constraints.

Output

Please output N lines, where the i-th line contains an integer indicating the loneliness of the i-th horse.

Sample Input 1

```
2
0 0
1 2
```

Sample Output 1

```
2
2
```

Sample Input 2

```
3
0 0
3 3
4 3
```

Sample Output 2

```
9
5
6
```

For any pair of horses you can only consider their differences in x-axis and y-axis, i.e., $|x_1 - x_2|$, $|y_1 - y_2|$. Let them be d_x and d_y respectively.

Because a "horse jump" is more efficient than "horse walk", we want to jump as more as possible. It can be proved that if $x_1 < x_2$ and $y_1 < y_2$, then using only $(+2, +1)$ and $(+1, +2)$ does not lose the optimality.

A greedy algorithm is:

```
ans = 0
while d_x > 0 and d_y > 0:
    if d_x > d_y:
        d_x -= 2
        d_y -= 1
    else:
        d_x -= 1
        d+y -= 2
    ans += 2

// one of d_x and d_y is 0
ans += d_x + d_y
```

The time complexity is $O(X)$ for each pair of horses where X indicating the range of the coordinates.

To improve this, you need to find that if $d_x = d_y$ during the process, then you will alternatively choose 2 kinds of horse jump. You can find the time needed to reach $d_x = d_y$ state (or know that this will never happens) in $O(1)$ time. When $d_x = d_y$, you can consider that the horse are doing some $(+3, +3)$ jumps and then you can find the answer in $O(1)$.

The lemma below shows that this algorithm is correct:

For a given d_x and d_y, the minimum time needed is

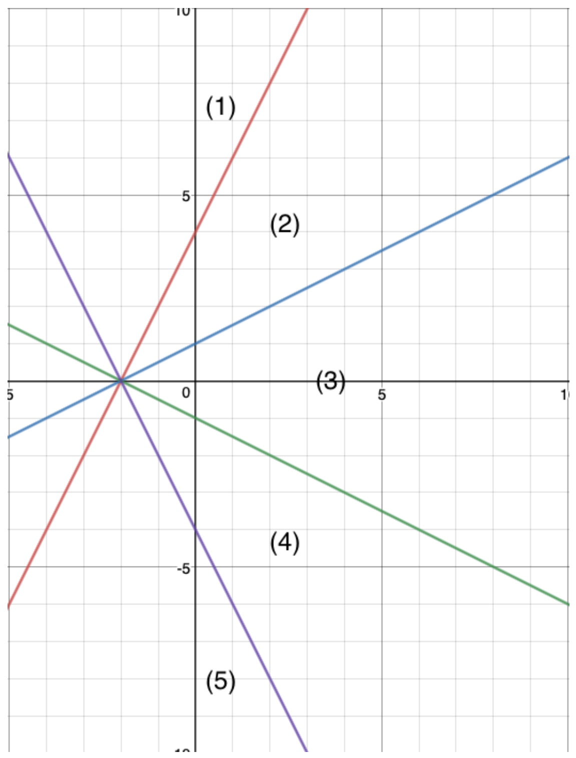
$$\begin{cases} d_x & \text{if } d_x > 2d_y \\ d_y & \text{if } d_y > 2d_x \\ 2\lfloor \frac{d_x+d_y}{3} \rfloor + ((d_x+d_y) \text{ modulo } 3) & \textit{otherwise} \end{cases}$$

The proof for the lemma is omitted.

If $d_x = d_y$ happens, it is the third case, and the algorithm will gives out the same answer with the lemma.

As for the bonus subtask, the solution from problem setter is divide and conquer. First divide all the points into 2 halves according to their x-axis, just like "Rectangle Area" in homework 1. In Rectangle Area, we know that for each left-hand-side point p, we can find the information of all the right-hand-side points below (above) a line passing p in amortized $O(1)$ time, i.e. find all N answers in $O(N)$ time.

Then for each point in the left, we can divide the right-hand-side points in 5 cases:



Case (1) and (5) need their sum of y axis, and this can be done by similar procedure in homework 1.

Case (3) need their sum of x axis, this can be done by Inclusion–exclusion principle. That is, find the sum of all the x axis in right-hand-side, and then subtract them by (1)(2) (the sum above some line) and (4)(5).

As for case (2), by

$$|x_2 - x_1| + |y_2 - y_1| = (x_2 + y_2) - (x_1 + y_1),$$

we need their sum of $x_i + y_i$. But because you need the quotient and remainder of their difference, you need to store them according to their remainder respect to 3. That is, you'll need to know how many points in (2) area that $x_i + y_i$ modulo 3 is 0, and what is their sum of $x_i + y_i$. The case for remainder is 1 or 2 is similar.

Case (4) is similar to case (2), and this needs the information of $x_i - y_i$.

Therefore we can find an $O(N \log^2 N)$ algorithm for this problem.

Here is the code for 100 points: http://codepad.org/JHMQ6bL3

Here is the code for 120 points: http://codepad.org/z3thoKxv

Problem 3-1 - Illuminati Matrix (Programming) (10 points)

Problem Description

Joe is convinced of Illuminati. He strongly believes that some numbers represent the holy trinity and calls them Illuminati numbers. We say a number x to be Illuminati number if it satisfies all of the following constraints:

- $x \equiv 0 \ (mod\, 33)$
- The number of digits 3 in x $\equiv 0 \ (mod\, 3)$
- The number of digits 6 in x $\equiv 0 \ (mod\, 3)$
- The number of digits 9 in x $\equiv 0 \ (mod\, 3)$

After you lost Joe in the stone game a few weeks ago, you got trapped in a matrix maze consisting of $(n \times m)$ cells. Each cell has a magic power. If you pass through a cell whose magic power satisfies the Illuminati constraints, you will be killed by Satan and be in hell forever.

Initially, you are in the top-leftmost cell, and the exit is in the bottom-rightmost cell. You want to know the number of possible paths to escape the maze, but you can only go right or down in order to leave the maze as fast as possible.

Input

The first line contains two integers n, m indicating the size of the matrix, where $1 \leq n, m \leq 1,000$.

For the next n lines, the i^{th} line contains m integers $A_{i,j}$ indicating the magic power in cell (i, j), where $1 \leq A_{i,j} \leq 10^{18}$.

Subtask 1 (20 %)

- $n, m \leq 4$

Subtask 2 (80 %)

- No other constraints.

Output

Output the number of safe simple paths from top-left corner to bottom-right corner module 10^9+7.

Sample Input 1

```
3 3
142 3 528
112 875 2475
615 781 1
```

Sample Output 1

```
3
```

Sample Input 2

```
2 5
1 666666 3 4 55
2 4 5 333333 1
```

Sample Output 2

```
0
```

For the second sample, even though "1 2 4 5 3 4 55 1" do not contain any Illuminati number, it is not a valid path since you can only go right or down.

We can simply solve this problem by 2-dimension DP. Let $DP_{i,j}$ be the number of paths from the left-up corner to the $ceil_{i,j}$.

```
if A[i][j] is a Illuminati number:
    DP[i][j] = 0
else:
    DP[i][j] = DP[i - 1][j] + DP[i][j - 1]
```

To check if a number x is a Illuminati number, we can convert the x to a string and calculate the frequency of each digit.

AC code: http://codepad.org/5c3etGjL

Problem 3-2 - Illuminati and Joe (Programming) (15 points)

Problem Description

After you escaped from the maze, Joe asked you to join the Illuminati. Your beliefs are the Pyramid, the Eye, the Light, and the Eternal Circle. You also know that the selfish pursuit of money is a hollow goal, but the pursuit of the goodness that money can create is one of humanity's greatest responsibilities. Therefore, you decided to devote yourself to Illuminati.

Now, you are doing your first job "strengthening the matrix maze". You want to know how many Illuminati numbers are there in a specific range, so that you can design a robust maze.

Input

There is only one line containing two integers l, r indicating the range, where $1 \leq l \leq r \leq 10^{1000}$.

Output

Output the number of Illuminati numbers in the range $[l, r]$ module $10^9 + 7$.

Subtask 1 (20 %)

- $l, r \leq 10^7$

Subtask 2 (80 %)

- No other constraints.

Sample Input

```
1 1243567
```

Sample Output

```
5245
```

Hint

Note that $(a + b) \mod x = ((a \mod x) + (b \mod x)) \mod x$

You may read the solution of problem 4 first, then you would probably know how to solve this problem.

The only difference is we have to record more states to do the dynamic programming.

$S(n, k, m, a, b, c)$ = { x | x is legal, of length n, begins with k, $x \equiv m$ (mod 33),
the number of digits 3 in x $\equiv a$ (mod 3),
the number of digits 6 in x $\equiv b$ (mod 3),
the number of digits 9 in x $\equiv c$ (mod 3)
}

AC code: http://codepad.org/s9LxG63c

Problem 4 - Digit Dynamic Programming (Hand-Written) (15 points)

Digit Dynamic Programming is a useful trick for solving problems with some constraints about digits. Let's take the following problem as an example.

Censorship

You have heard of *ADA Kingdom* in homework 1, but I think few of you know the miserable history of this kingdom. Let me tell you a story.

ADA Kingdom is actually built by *Handsome*, a TA of ADA course, by using his outstanding appearance and charisma. However, "power tends to corrupt and absolute power corrupts absolutely." Only in a few months, *ADA Kingdom* had become an autocratic country under *Handsome*'s terror reign.

In order to consolidate his power, he decided to eliminate all *anti-handsome* stuff, that is, ugly stuffs. Some 2-digit numbers, for some reasons, are considered *ugly*. In addition, every integer that contains any ugly number as its substring, when written in decimal, is considered *illegal*. For example, if both "87" and "38" are ugly, then "**87**", "1**387**" and "1**87**63" are all considered illegal, while "378" and "30083" are legal.

You are a mathematician in *ADA Kingdom*, and you would like to know how many available integers are there in a specified interval. Now, given those ugly 2-digit numbers, and an integer N, how many legal numbers are there between 1 and N? Please come up with an algorithm to compute this, with time complexity $O(\lg N)$.

Note that when input and output, the length of N, in decimal, is only $O(\lg N)$. Thus, neither inputting N nor outputting the answer would cause `TLE` directly.

The time limit is too tight to enumerate all the legal/illegal integers. Fortunately, we can compute this through simple dynamic programming.

For simplicity, a function S and an array `dp` are defined as follow.

$$S(n,k) = \{x | x \text{ is legal, of length } n\text{, and begins with } k\text{. If } k = 0 \text{ then } x \text{ can have leading zeros.}\}$$

$$\texttt{dp[n][k]} := |S(n,k)|.$$

For example, "378" belongs to $S(3,3)$, and "30083" belongs to $S(5,3)$.

Since the length of N is only $O(\lg N)$, and all possible value of k is between 0 and 9, the size of the `dp` array can only be $10(\lg N + 1)$, which is in $O(\lg N)$

(1) (5 pts) Obviously, every legal number $a_1a_2...a_n$ can be viewed as a_1 followed by a legal number $a_2a_3...a_n$. In the other hand, for every legal number $a_2a_3...a_n$, if a_1a_2 is not prohibited, then $a_1a_2...a_n$ is also a legal number.

Based on the observation above, and assuming that every `dp[i][j]` where `i` $<$ `n`, has been computed. Please describe how to compute `dp[n][k]` in $O(1)$.

```
dp[n][k]=0
for i from 0 to 9:
    if 10k+i is not ugly:
        dp[n][k]+=dp[n-1][i]
```

(2) (2 pts) A legal integer should not begin with zero. Then, do we still need `dp[n][0]`? Why or why not? (The answer depends on your implementation for problem (1) and (3).)

Yes. When calculating `dp[n][k]`, the second digit can be 0, which leads to the requirement of `dp[n-1][0]`.

(3) (8 pts) Now assume that you've computed the whole `dp` array. Please design an algorithm to compute the number of legal numbers in $[0, N)$ with time complexity $O(\lg N)$.

Assume that the length of N is `l`, and the decimal representation of N is `a[l]a[l-1]a[l-2]...a[1]`.

```
ans := 0
ugl := false
for n from l to 1:
    if ugl is false:
        for k from 0 to a[n]-1:
            if n==l or 10a[n+1]+k is not ugly:
                ans += a[n][k]
        if n < l and 10a[n+1]+a[n] is ugly:
            ugl = true
```

Problem 5 - The Robbers (Hand-Written) (30 points)

In 2500 A.D., the world suffers from drugs, violence, and crimes.

It is no longer a peaceful, but horrible and suspicious society. The gap between the rich and the poor is becoming larger and larger. Moreover, because the population density is much higher than before, the living space is extremely compressed and packed. As a result, the roads become considerably narrow, and in turn makes robbery much easier to rob people than before. Under such circumstances, robbers can easily find a victim and rob all of his/her valuable belongings, because the victims have no way to escape. Robbers, accordingly, have become one of the most thriving occupations in this evil society. What's worse, the robbers are often brilliant people, so they always work as a group and use their intelligence to rob as much as possible. Consequently, there is almost no way for an innocent people to escape from their threat. They have no choice but to give out their belongings. To the robbers, taking over one's valuable belongings is only a matter of time.

(1) There are N innocent people standing in a line on one side of a narrow one-end alley, and M ($\leq N$) robbers in a group standing in a line on the other side of the alley. Note that the alley is extremely narrow so that the innocent people cannot exchange the relative positions with others. In addition, a security, who is already bribed by the robbers, has closed the only entrance of the alley, so the people can never escape.

Some people are so strong that they can resist for a longer period of time, while others are not powerful enough to fight with the robbers. Namely, the i-th person can resist for t_i ($0 < t_i \leq N$) seconds, which implies that a robber needs t_i seconds to acquire all the belongings of him/her. Moreover, one robber can only rob one person at a time, so if a robber would like to rob the first and second people, then it will take him $t_1 + t_2$ seconds.

The robbers are going to rob all the people in the alley. Since they cannot change their relative positions, one single robber can only rob a consecutive interval of people, and all the intervals cannot overlap with one another. Also, because the robbers work as a group, the overall time the group spends on robbing all the people will be the maximum time spent by individual members. For example, if there are 3 robbers spending $5, 7, 9$ seconds robbing people respectively, then the overall time spent is 9 seconds.

(a) (4 pts) The police are going to arrive at the alley in T seconds. Therefore, the robbers should finish their crime before the police arrive. Given the value of T, please design a greedy algorithm to decide whether the robbers can rob all the innocent people before the police arrive. Your algorithm should run in $O(N)$ time, and you should justify why your algorithm satisfies the time requirement.

We would like to let the left-most robber to start robbing the left-most innocent people since he is the only one robber who can rob him. Next, within the time limit T, we would like to let him rob as many people as he can because it will alleviate the workload of remaining robbers. Please refer to the following pseudo-code. The time complexity is obviously $O(N)$ because there is only one for-loop traversing all t_i, and no other traversal or process is done.

```
solve(T):
    sum = 0, ans = 0
    for i in {1..N}
        if t_i > T
            return False
        sum += t_i
        if sum > T
            ans += 1
            sum = t_i
    if ans > M
        return False
    else return True
```

(b) (5 pts) Please show the correctness of your algorithm in (a) by proving that it satisfies *Greedy Choice Property* and *Optimal Substructure*.

Note that there are multiple approaches to prove these properties. Any valid proof will be considered correct.

Greedy Choice Property:

Consider a solution provided by the greedy algorithm in (a) is not optimal. That is, there exists another solution implying that the number of robber needed is less than `ans`. Note that such proof will be sufficient since in the original problem, M is only a threshold. Therefore, if we prove that (a) can always find the minimum number of robbers needed, then apparently (a) will be no worse than other solutions, which implies that the property can be proved. Let OPT be the optimal strategy.

Case 1: In OPT, the first robber robs exactly the same number of people as (a) has suggested. $\Rightarrow$ Greedy Choice is included in OPT.

Case 2: In OPT, the first robber robs more number of people than (a) has suggested. $\Rightarrow$ Contradiction, since in (a), the suggested number of innocent people is already the maximum. If there exists a strategy such that the first robber robs more people than (a) suggests, then (a) should let the first robber rob more people. Therefore, such case does not exist.

Case 3: In OPT, the first robber robs less number of people than (a) has suggested. Under such circumstance, we can always let the robber to rob more people instead, and the robbers needed will become equal or more than the original OPT since the number of innocent people remaining is decreased. Hence, (a) is at least not worse than OPT.

From Case 1-3, we can say that OPT must include the decision made in (a). That is, the algorithm in (a) satisfies Greedy Choice Property.

Optimal Substructure:

Let x to be the maximum number of innocent people that the first robber can rob within T seconds.

If we let the first robber rob i people, where $0 \leq i \leq x$, then, we have $M - 1$ robbers and $N - i$ innocent people left. If under such conditions, the remaining robbers can rob all of them within T seconds, then overall the M robbers can finish their jobs within T seconds, and vice versa. Hence, such problem has optimal structure.

(c) (2 pts) Let $f(T)$ be a function where T is a non-negative integer. The value of $f(T)$ is 1 if the robbers can finish their jobs within T seconds, and 0 if the robbers cannot finish. Show that $f(T)$ is a monotonically increasing (i.e. non-decreasing) function.

We want to prove that $f(T_2) - f(T_1) \geq 0, \forall T_2 > T_1 \geq 0$

Proof:

1. When $f(T_1) = 0$, then $f(T_2) - f(T_1) \geq 0$ is obviously correct since the $f(T_2)$ can only be 0 or 1.

2. When $f(T_1) = 1$, by definition, robbers can finish their job within T_1 seconds. Since $T_2 > T_1$, robbers can definitely finish their job within T_2 seconds, which implies $f(T_2) = 1$. Hence, $f(T_2) - f(T_1) = 0 \geq 0$

By 1., 2., and the fact that the range of f is $\{0, 1\}$, we can say that f is a monotonically increasing function.

(d) (2 pts) Given that $N = 10$, $M = 3$, and t_i as follows, please propose a plan (i.e., how should the M robbers divide their work) and calculate the shortest possible time for the robbers to rob all of the innocent people.

$$5, 8, 2, 1, 6, 4, 10, 9, 7, 3$$

The shortest possible time is 20, with 3 robbers robbing [1, 4], [5, 7], [8, 10] -th people respectively.

(e) (5 pts) Please design an algorithm to calculate the minimum time the robbers need to rob all the people. Your algorithm should run in $o(N^2)$ time. Note that the complexity is small-o, not big-o. Please also justify that your algorithm meets the time complexity requirement.

From (c), we know that f is a monotonically increasing function, and the minimum possible time for robbers to finish their job is the minimum T_{ans} such that $f(T_{ans}) = 1$. Therefore, we can use **binary search** to search for T_{ans}. The upper bound of T_{ans} will be Σt_i, and since $t_i \leq N$, we can revise the upper bound to be N^2. Please refer to the following pseudo code:

```
lbound = 0, rbound = N^2 + 1 // range = [lbound, rbound)
while rbound - lbound > 1
    mid = (rbound + lbound) / 2
    if solve(mid)
        rbound = mid + 1
    else
        lbound = mid + 1
return lbound
```

The function `solve` takes $O(N)$ time as suggested in (a), and it takes at most $\log_2(N^2)$ attempts to converge the possible range. Therefore, the overall time complexity is $O(N \lg N)$, which satisfies the constraint.

(2) After robbing all the people inside the lane, the leader of those robbers shouted "*Hey, the shopkeeper there, our bag can still hold some stuffs of total weight W!*"

You, a shopkeeper at the end of the lane, are still curious about why they gave you such information, but you soon realize that you'd better lock some of your goods into your vault, because the robbers will rob you in minutes!

You have N goods, numbered as 1, 2, ..., N. The i-th of them is of weight w_i and of value v_i. Due of the size limit of the vault, you can only put K ($< N$) of them into the vault. After the robber come, they will choose some goods from the rest ($N - K$) goods, which are not in the vault. Since they are experienced robbers, they will rob in the optimal way. That is, they will choose some of the remaining goods to maximize the total value V, with total weight not greater than W, just like the well-known knapsack problem. Now, given N, K, W, w_1, w_2, ..., w_N, v_1, v_2, ..., and v_N.

(a) (4 pts) As a shopkeeper, your goal is to minimize V, the total robbed value.

First, let's focus on the special case when $K = 1$. That is, you can only put one good into the vault.

By solving a knapsack problem with backtracking on the N goods, you've computed a set of goods, S, such that taking S away is one of the optimal plans for the robbers when you don't put anything into the vault.

Observe that when $K = 1$, without loss of optimality, the good you need to lock must be in S.

Please design an algorithm with time complexity $O(|S|NW)$ to calculate the minimized V for the special case $K = 1$.

Let $G = \{g_1, g_2, ..., g_N\}$ be the set of those goods, and $S \subset \{1, 2, ..., N\}$.

```
V:=sum of v[i]
for s in S:
    G' := G\{s}
    V = min(V, Knapsack(G', W) )
Output V
```

(b) (8 pts) (You can still get 4 pts in the problem even if you didn't answer it, and additional 4 pts will be given if your answer to the revised problem is correct.)

Soon, you changed your mind. Because all those goods are insured, to let them robbed can turn them into money! Since your shop has only little business, it seems that it is the only way to turn those goods into money. As a result, you decided to maximize V. However, to prevent the incident to be seen as an insurance fraud, you still need to put exactly K items into the vault. Please design an algorithm to calculate the maximized V in $O((N + W)^3)$ time for the general case.

```
for n from 0 to N:
    for l from 0 to W:
        for c from 0 to N-K:
            dp[n][l][c]=0
            if n>0:
                dp[n][l][c]=dp[n-1][l][c]
                if l >= w[n] and c > 0:
                    dp[n][l][c]=max(dp[n][l][c],
                                        dp[n-1][l-w[n]][c-1]+v[n])
Output dp[N][W][N-K]
```

Problem 6 - The ADA Chocolate Factory (Hand-Written) (12 points)

Zolution, a TA of ADA, is also a manager of a chocolate company named `ADA Chocolate`. His secret recipe, *love*, soon made his company become the world's biggest chocolate company. `ADA Chocolate` is well known for its perfect square chocolate, which is composed of $N \times N$ cells.

Due to greed, a vicious TA *Handsome* established another chocolate company `AD4 Ch0co1ate` to compete with him. With all dirty tricks, such as smear, hacking, and faking, *Handsome*'s company caught up with *Zolution*'s, and became one of the top three chocolate companies.

Being aware of the competitive threat of *Handsome*'s business, *Zolution* decided to build a brand-new, cutting-edge chocolate factory with high technology and pioneering infrastructure. However, building such a new factory takes time and money, and in the meanwhile *Zolution* wants to expand his profits as soon as possible. Therefore, he would like to maximize the profits earned when the new factory is still under construction.

There are N necessary components $x_1, x_2, ...x_N$ to be constructed and installed in the new factory. Each component can increase the output of the factory. The more components are installed, the higher output the factory can achieve. Yet, the components cannot be built simultaneously. The i-th component will take you t_i full days to construct, and will make the factory produce v_i more chocolates every day after the installation. Once a component is constructed, it will be immediately installed at the end of the day. Initially, the factory cannot produce any chocolate.

(1) (2 pts) Consider that there are only two components x_1, x_2, whose time cost and extra chocolate productivity are t_1, t_2 and v_1, v_2 respectively. Which component will you prefer to build first? Briefly explain your reason.

Case 1: x_1 is built first $\Rightarrow$ The profit will be $v_1 \times t_2$

Case 2: x_2 is built first $\Rightarrow$ The profit will be $v_2 \times t_1$

Therefore, if $v_1 \times t_2 > v_2 \times t_1$, building x_1 first will be more profitable, or building x_2 first will be better.

(2) (5 pts) Please design an algorithm to calculate the maximum total number of chocolate produced from day 1 to day Σt_i. Your algorithm should run in $O(n \log n)$. You need to explain why your algorithm meets the time complexity requirement.

$v_1 \times t_2 > v_2 \times t_1 \Rightarrow v_1/t_1 > v_2/t_2$

Hence, we sort the components $x_1, x_2, ..., x_N$ by their values of v_i/t_i, and build the components greedily from the largest v_i/t_i to the smallest. The time complexity will be $O(N \log N)$ based on sorting algorithm, which satisfies the requirement.

(3) (5 pts) Please prove the correctness of your proposed algorithm in (2). Note that if you use Greedy or Dynamic Programming, you should also prove their properties.

Note that there are multiple approaches to prove these properties. Any valid proof will be considered correct.

Without loss of generality, in the following proof, $\{x_1, x_2, ...x_N\}$ is an sorted sequence according to their v_i/t_i values from largest to smallest.

Greedy Choice Property:

Assume the strategy in (2) is not optimal, then the optimal strategy OPT should be with building sequence $\{x_{a_1}, x_{a_2}, ..., x_{a_N}\}$, and in the sequence, exists some i such that $a_i > a_{i+1}$. (If such i does not exist, then it is obvious that OPT still has a sorted building sequence, and exchange those two components with same v_i/t_i will not affect the profit. Therefore, through exchange, OPT can be modified to the same strategy with (2))

The overall profit of OPT will be $(T - t_{a_1}) \times v_{a_1} + (T - t_{a_1} - t_{a_2}) \times v_{a_2} + ...$, where $T = \Sigma t_i$. The formula is equivalent to $\Sigma_{i=1}^{n}((\Sigma_{j=i+1}^{N} t_{a_j}) \times v_{a_i})$. If we exchange a_i with a_{i+1}, the profit difference will be $v_1 \times t_2 - v_2 \times t_1$. Moreover, since $a_i > a_{i+1} \Rightarrow v_{a_i}/t_{a_i} > v_{a_{i+1}}/t_{a_{i+1}}$, the profit will increase after the exchange, which implies that OPT is NOT an optimal solution. Hence, there is a contradiction, and it implies that the strategy in (3) should be optimal. $\Rightarrow$ Greedy Choice Property is proved.

Optimal Substructure:

Let OPT be the optimal solution to build components $\{x_i, x_{i+1}, ..., x_N\}$.

Case 1: OPT builds x_1 first:

$OPT \backslash \{x_1\}$ will be the optimal solution to build components $\{x_1, x_2, ..., x_N\} \backslash x_1$ since no matter how were rest of the components built, the profit made by x_1 is always $\Sigma_{i=2}^{N} t_i \times v_1$. Therefore, $OPT \backslash \{x_1\}$ should be the optimal solution to make OPT an optimal solution.

Case 2: OPT builds x_2 first to Case N: OPT builds x_N first can all be proved by the same methods. Hence, the problem has an optimal substructure.