CSCI 269 – Fall 2014 Handout: Dynamic Programming
Computational Problem Solving
Michael H. Goldwasser
Saint Louis University Tuesday, 16 September 2014

1 Dynamic Programming

For today's practice, the theme is Dynamic Programming, which is a technique that can be used to provide efficient solutions for a number of optimization or feasibility problems that have a natural divide-and-conquer structure. As you will see from the success rates in the table below, these problems are more difficult than the average problem, but they are solvable (especially by the teams competing for the top ranks).

Although it is sometimes a challenge to figure out how to solve a problem using dynamic programming, the good news is that its often relatively easy to code a solution once you understand how it works.

The judge's data for all of these problems are loaded onto turing so that you can use our submit script to test your solutions.
(See `http://cs.slu.edu/~goldwasser/icpc/resources/` for details and `http://cs.slu.edu/~goldwasser/cgi-bin/icpc` for links to problem statements.)

shortname	Problem	contest	success	notes
robot	Robot Challenge	2009 Southeast	31%	
ripoff	RIPOFF	2009 Mid-Central	14%	
pascal	Pascal's Travels	2005 Mid-Central	12%	
mosaic	Mosaic	2009 Southeast	3.7%	tiling
shut	Shut The Box	2011 Mid-Central	1.4%	bit manip
lawrence	Lawrence of Arabia	2008 Rocky Mountain	0%	
bracelet	Bright Bracelet	2003 Mid-Central	??	bit manip
//moving	Adventures in Moving – Part IV	2001 Waterloo Local	??	

H: Robot Challenge

You have entered a robot in a Robot Challenge. A course is set up in a 100m by 100m space. Certain points are identified within the space as targets. They are ordered – there is a target 1, a target 2, etc. Your robot must start at (0,0). From there, it should go to target 1, stop for 1 second, go to target 2, stop for 1 second, and so on. It must finally end up at, and stop for a second on, (100,100).

Each target except (0,0) and (100,100) has a time penalty for missing it. So, if your robot went straight from target 1 to target 3, skipping target 2, it would incur target 2's penalty. Note that once it hits target 3, it cannot go back to target 2. It must hit the targets in order. Since your robot must stop for 1 second on each target point, it is not in danger of hitting a target accidentally too soon. For example, if target point 3 lies directly between target points 1 and 2, your robot can go straight from 1 to 2, right over 3, without stopping. Since it didn't stop, the judges will not mistakenly think that it hit target 3 too soon, so they won't assess target 2's penalty. Your final score is the amount of time (in seconds) your robot takes to reach (100,100), completing the course, plus all penalties. Smaller scores are better.

Your robot is very maneuverable, but a bit slow. It moves at 1 m/s, but can turn very quickly. During the 1 second it stops on a target point, it can easily turn to face the next target point. Thus, it can always move in a straight line between target points.

Because your robot is a bit slow, it might be advantageous to skip some targets, and incur their penalty, rather than actually maneuvering to them. Given a description of a course, determine your robot's best (lowest) possible score.

The Input

There will be several test cases. Each test case will begin with a line with one integer, N ($1 \le N \le 1000$) which is the number of targets on the course. Each of the next N lines will describe a target with three integers, X, Y and P, where (X,Y) is a location on the course ($1 \le X,Y \le 99$, X and Y in meters) and P is the penalty incurred if the robot misses that target ($1 \le P \le 100$). The targets will be given in order – the first line after N is target 1, the next is target 2, and so on. All the targets on a given course will be unique – there will be at most one target point at any location on the course. End of input will be marked by a line with a single `0`.

The Output

For each test case, output a single decimal number, indicating the smallest possible score for that course. Output this number rounded (NOT truncated) to three decimal places. Print each answer on its own line, and do not print any blank lines between answers.

Sample Input

```
1
50 50 20
3
30 30 90
60 60 80
10 90 100
3
30 30 90
60 60 80
10 90 10
0
```

Sample Output

```
143.421
237.716
154.421
```

Problem I: RIPOFF

Source file: `ripoff.{c, cpp, java}`
Input file: `ripoff.in`

Business has been slow at Gleamin' Lemon Used Auto Sales. In an effort to bring in new customers, management has created the Rebate Incentive Program Of Fabulous Fun (or RIPOFF). This is a simple game which allows customers to try and win a rebate on an automobile purchase. The RIPOFF game is a board game where each square is labeled with a rebate amount. The customer advances through the board by spinning a spinner. Each square he lands on adds to his total rebate amount. When he reaches the end of the board he is rewarded with the total rebate amount.

Of course, given the company involved, it should come as no surprise that there are a couple of catches written in the fine print. The first is that there is a limit to the number of turns the customer has to finish the game; if he doesn't reach the end within the allotted number of turns then he loses his rebate. The second is that some of the squares actually have a negative amount which subtract from the rebate instead of adding to it. A particularly unlucky customer might even come out of the game with a negative rebate.

Even with these catches, the management of Gleamin' Lemon is concerned that someone might win a particularly large rebate—something they would like to avoid at all costs. Your job is to take a particular configuration for the RIPOFF game and decide the maximum rebate a customer could possibly obtain.

Consider, for example, the game board below. Assume we have 5 turns to finish the game, and each turn we can move between 1 and 4 spaces depending on what we spin. Notice that we must start just before the board begins, so spinning a 1 causes us to land on the first square. Also notice we must end by landing past the end of the last square. It does not have to be exact; any number that gets us off of the board will work.

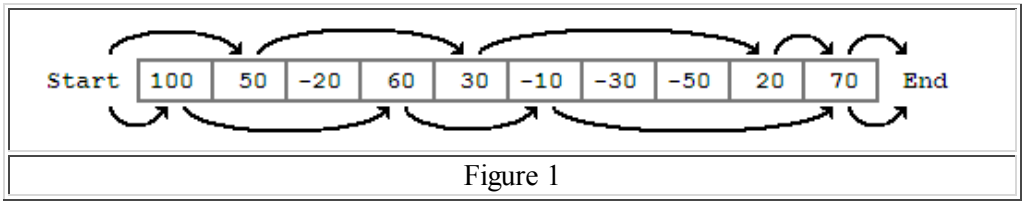


Figure 1

The illustration shows two different possible ways the game might go. Following the arrows on the top, if we spin a 2, 3, 4, 1, and 1 respectively, we will win a total rebate of 50 + 30 + 20 + 70 = $170. However, the best possible rebate we could win would be $220. We would win this amount if we spun a 1, 3, 2, 4, and 1 respectively, as shown by the lower path. Notice that we did not land on every square with a positive number; if we had we wouldn't have been able to make it to the end of the board before the 5 turns was up.

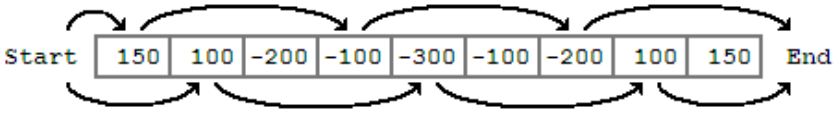


Figure 2

The illustration in Figure 2 shows a game where we have 4 turns to finish the game, and can move up to 3 spaces each turn. Again, two different paths are shown, the one on top earning a rebate of -$150, and the one on bottom earning a rebate of -$100. In fact, -$100 is the highest possible rebate we could earn for this game (a fact that would no doubt please the management of Gleamin' Lemon). Of course, there also might be a sequence of moves in which we do not reach the end before the turn limit—e.g. spinning a 1 every time. Although not finishing would actually be preferable to finishing with a negative rebate, in this problem we are only going to consider sequences of moves which allow us to reach the end before the turn limit.

Input: The input consists of one to twenty data sets, followed by a line containing only 0.

The first line of a data set contains three space separated integers N S T, where

N is the total number of squares on the board, $2 \le N \le 200$.
S is the maximum number of spaces you may advance in each turn, $2 \le S \le 10$.
T is the maximum number of turns allowed, where $N + 1 \le ST$ and $T \le N + 1$.

The data set ends with one or more lines containing a total of N integers, the numbers on the board. Each number has magnitude less than 10000.

Output: The output for each data set is one line containing only the maximum possible rebate that can be earned by completing the game.

To complete the game you must advance a total of $N + 1$ spaces in at most T turns, each turn advancing from 1 to S spaces inclusive. It will always be possible to complete a game. However, there may be a very large number of different turn sequences that will finish, so you will need to be careful in choosing your algorithm.

The sample input data corresponds to the games in the Figures.

Example input:	**Example output:**
10 4 5 100 50 -20 60 30 -10 -30 -50 20 70 9 3 4 150 100 -200 -100 -300 -100 -200 100 150 0	220 -100

Last modified on October 18, 2009 at 9:58 AM.

Problem A: Pascal's Travels

Source file: `pascal.{c, cpp, java}`
Input file: `pascal.in`

An *n* x *n* game board is populated with integers, one nonnegative integer per square. The goal is to travel along any legitimate path from the upper left corner to the lower right corner of the board. The integer in any one square dictates how large a step away from that location must be. If the step size would advance travel off the game board, then a step in that particular direction is forbidden. All steps must be either to the right or toward the bottom. Note that a 0 is a dead end which prevents any further progress.

Consider the 4 x 4 board shown in Figure 1, where the solid circle identifies the start position and the dashed circle identifies the target. Figure 2 shows the three paths from the start to the target, with the irrelevant numbers in each removed.

2	3	3	1
1	2	1	3
1	2	3	1
3	1	1	0

Figure 1

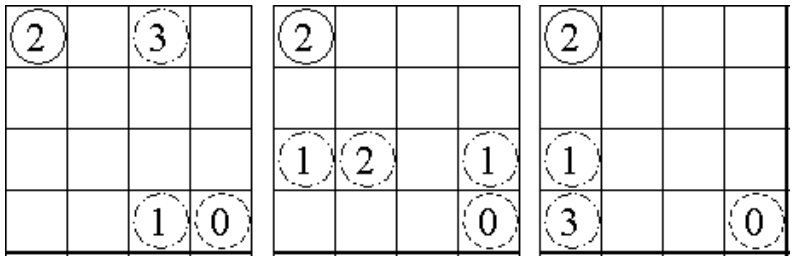


Figure 2

Input: The input contains data for one to thirty boards, followed by a final line containing only the integer -1. The data for a board starts with a line containing a single positive integer *n*, $4 \le n \le 34$, which is the number of rows in this board. This is followed by *n* rows of data. Each row contains *n* single digits, 0-9, with no spaces between them.

Output: The output consists of one line for each board, containing a single integer, which is the number of paths from the upper left corner to the lower right corner. There will be fewer than 2^{63} paths for any board.

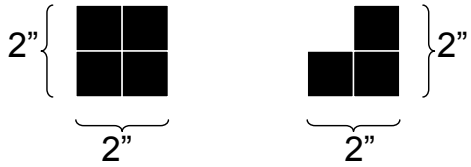
Warning: Brute force methods examining every path will likely exceed the allotted time limit. 64-bit integer values are available as *long* values in Java or *long long* values using the contest's C/C++ compilers.

Example input:	Example output:
4 2331 1213 1231 3110 4 3332 1213 1232 2120 5 11101 01111 11111 11101 11101 -1	3 0 7

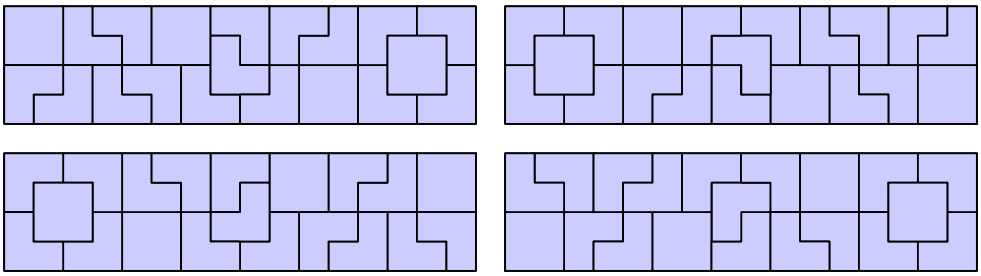
Last modified on October 26, 2005 at 6:45 PM.

I: Mosaic

An architect wants to decorate one of his buildings with a long, thin mosaic. He has two kinds of tiles available to him, each 2 inches by 2 inches:



He can rotate the second kind of tile in any of four ways. He wants to fill the entire space with tiles, leaving no untiled gaps. Now, he wonders how many different patterns he can form. He considers two mosaics to be the same only if they have exactly the same kinds of tiles in exactly the same positions. So, if a rotation or a reflection of a pattern has tiles in different places than the original, he considers it a different pattern. The following are examples of 4” x 16” mosaics, and even though they are all rotations or reflections of each other, the architect considers them to be four different mosaics:



Input

There will be several test cases. Each test case will consist of two integers on a single line, `N` and `M` ($2 \le N \le 10$, $2 \le M \le 500$). These represent the dimensions of the strip he wishes to fill, in inches. The data set will conclude with a line with two 0’s.

Output

For each test case, print out a single integer representing the number of unique ways that the architect can tile the space, **modulo 10^6**. Print each integer on its own line, with no extra whitespace. Do not print any blank lines between answers.

Sample Input

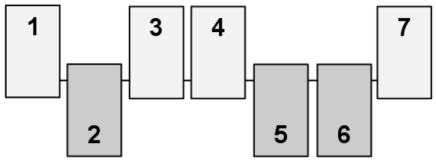
```
2 10
4 16
4 50
0 0
```

Sample Output

```
25
366318
574777
```

Problem F: Shut the Box

Source file: `shut.{c, cpp, java}`
Input file: `shut.in`



Shut the Box is a one-player game that begins with a set of *N* pieces labeled from 1 to *N*. All pieces are initially "unmarked" (in the picture at right, the unmarked pieces are those in an upward position). In the version we consider, a player is allowed up to *T* turns, with each turn defined by an independently chosen value *V* (typically determined by rolling one or more dice). During a turn, the player must designate a set of currently *unmarked* pieces whose numeric labels add precisely to *V*, and mark them. The game continues either until the player runs out of turns, or until a single turn when it becomes impossible to find a set of unmarked pieces summing to the designated value *V* (in which case it and all further turns are forfeited). The goal is to mark as many pieces as possible; marking all pieces is known as "shutting the box." Your goal is to determine the maximum number of pieces that can be marked by a fixed sequence of turns.

As an example, consider a game with 6 pieces and the following sequence of turns: 10, 3, 4, 2. The best outcome for that sequence is to mark a total of four pieces. This can be achieved by using the value 10 to mark the pieces 1+4+5, and then using the value of 3 to mark piece 3. At that point, the game would end as there is no way to precisely use the turn with value 4 (the final turn of value 2 must be forfeited as well). An alternate strategy for achieving the same number of marked pieces would be to use the value 10 to mark four pieces 1+2+3+4, with the game ending on the turn with value 3. But there does not exist any way to mark five or more pieces with that sequence.

Hint: avoid enormous arrays or lists, if possible.

Input: Each game begins with a line containing two integers, *N*, *T* where $1 \le N \le 22$ represents the number of pieces, and $1 \le T \le N$ represents the maximum number of turns that will be allowed. The following line contains *T* integers designating the sequence of turn values for the game; each such value *V* will satisify $1 \le V \le 22$. You must read that entire sequence from the input, even though a particular game might end on an unsuccessful turn prior to the end of the sequence. The data set ends with a line containing 0 0.

Output: You should output a single line for each game, as shown below, reporting the ordinal for the game and the maximum number of pieces that can be marked during that game.

Example input:	**Example output:**

`6 4`
`10 3 4 2`
`6 5`
`10 2 4 5 3`
`10 10`
`1 1 3 4 5 6 7 8 9 10`
`22 22`
`22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1`
`0 0`
	`Game 1: 4`
	`Game 2: 6`
	`Game 3: 1`
	`Game 4: 22`

Problem B: Lawrence of Arabia

Source: `lawrence.{c,cpp,java}`
Input: `lawrence.in`
Output: `lawrence.out`

T. E. Lawrence was a controversial figure during World War I. He was a British officer who served in the Arabian theater and led a group of Arab nationals in guerilla strikes against the Ottoman Empire. His primary targets were the railroads. A highly fictionalized version of his exploits was presented in the blockbuster movie, "Lawrence of Arabia".

You are to write a program to help Lawrence figure out how to best use his limited resources. You have some information from British Intelligence. First, the rail line is completely linear---there are no branches, no spurs. Next, British Intelligence has assigned a Strategic Importance to each depot---an integer from 1 to 5. A depot is of no use on its own, it only has value if it is connected to other depots. The Strategic Value of the entire railroad is calculated by adding up the products of the Strategic Values for every pair of depots that are connected, directly or indirectly, by the rail line. Consider this railroad:

Its Strategic Value is 4*5 + 4*1 + 4*2 + 5*1 + 5*2 + 1*2 = 49.

Now, suppose that Lawrence only has enough resources for one attack. He cannot attack the depots themselves---they are too well defended. He must attack the rail line between depots, in the middle of the desert. Consider what would happen if Lawrence attacked this rail line right in the middle:

The Strategic Value of the remaining railroad is 4*5 + 1*2 = 22. But, suppose Lawrence attacks between the 4 and 5 depots:

The Strategic Value of the remaining railroad is 5*1 + 5*2 + 1*2 = 17. This is Lawrence's best option.

Given a description of a railroad and the number of attacks that Lawrence can perform, figure out the smallest Strategic Value that he can achieve for that railroad.

Input

There will be several data sets. Each data set will begin with a line with two integers, n and m. n is the number of depots on the railroad ($1 \leq n \leq 1000$), and m is the number of attacks Lawrence has resources for ($0 \leq m < n$). On the next line will be n integers, each from 1 to 5, indicating the Strategic Value of each depot in order. End of input will be marked by a line with $n=0$ and $m=0$, which should not be processed.

Output

For each data set, output a single integer, indicating the smallest Strategic Value for the railroad that Lawrence can achieve with his attacks. Output each integer in its own line.

Sample Input

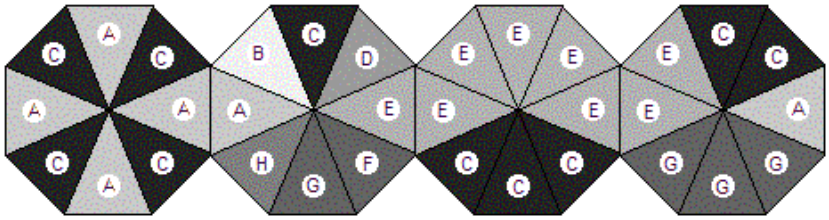
```
4 1
4 5 1 2
4 2
4 5 1 2
0 0
```

Sample Output

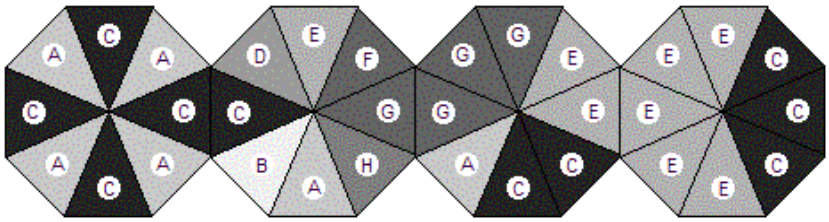
```
17
2
```

2784 - Bright Bracelet

North America - Mid Central - 2003/2004



Bracelet 1



Bracelet 2

Bracelets can be made from a collection of octagonal pieces, with two opposite sides of an octagon attached to octagons on either side. The colors of the edges of the octagons vary. The different colors are labeled with different letters in the diagrams. Bracelets only look good if the connecting sides of two adjacent octagons are the same color. Above are two possible bracelets. (The ends also get fastened together.) These two bracelets could be made from the same four octagons, reordered and rotated. Assume that the octagons are never flipped over.

It happens that the better selling bracelets are those with the darker colors on the edges connecting the bracelet. The brightness of each lettered color is a positive integer, with higher numbers being brighter. Suppose the brightness of the labeled colors are:

A	B	C	D	E	F	G	H
70	90	10	50	60	30	20	40

We can compare the desirability of these two arrangements of the octagons by adding the brightness of the colors at each joint (including the connection of the two ends). For Bracelet 1, colors A, A, E, and E have sum 70 + 70 + 60 + 60 = 260. For Bracelet 2, colors C, C, G, and E have sum 10 + 10 + 20 + 60 = 100. Bracelet 2 is preferable, having the smaller sum. In fact, Bracelet 2 provides the best possible result among all rearrangements and rotations of these four octagons.

Input

There are from one to 20 data sets, followed by a final line containing only 0. A data set starts with a line containing nine blank-separated integers. The first is the number, n, of octagons that form the bracelet, where $4 \le n \le 11$. The remaining eight numbers are the brightness for colors A through H, in order. Each brightness is positive and less than 256.

The next *n* lines each contain eight letters, all in the range from A through H. Each gives the edge colors for one octagon, in clockwise order. Individual colors may appear zero or more times in the octagons. Different colors may have the same brightness, but that does not make them the same color.

Output

The output contains one line for each data set: If no bracelet can be constructed using all the octagons, the line contains "impossible". Otherwise the line contains the minimal sum of the brightness for the connections. Caution: If your solution considers all possible orderings and rotations individually, it will run out of time.

Sample Input

```
4 70 90 10 50 60 30 20 40
ACACACAC
ABCDEFGH
EEEEECCC
EECCAGGG
5 1 2 3 4 5 6 7 8
AAAABBBB
BBBBCCCC
CCCCDDDD
DDDDEEEE
EEEEAAAA
6 50 50 50 50 100 1 2 3
HHHHHHHH
BBBBCCCC
CDCDDDDD
DEDEEEEE
EFEFEFEF
FFFFFFFF
0
```

Sample Output

```
100
15
impossible
```

Mid Central 2003-2004