



# Chapter 1 Practice for Ad Hoc Problems

Yonghui Wu

Shanghai Key Laboratory of Intelligent Information Processing

School of Computer Science, Fudan University

[yhwu@fudan.edu.cn](mailto:yhwu@fudan.edu.cn)



# Algorithm Design Practice

- This lecture is supported by Office of Global Partnerships (Key Projects Development Fund), Fudan University.

# Chapter 1 Practice for Ad Hoc Problems

- 1.1 Solving Problems by Mechanism Analysis
- 1.2 Solving Problems by Statistical Analysis



# Ad Hoc Problems

- **Ad hoc:** “for a special purpose .....”.
- No classical algorithms that can solve ad hoc problems. Programmers need to design specific algorithms to solve ad hoc problems.
- **Two strategies** to design algorithms for solving ad hoc problems:
  - **Mechanism analysis**
  - **Statistical analysis.**

# 1.1 Solving Problems by Mechanism Analysis

- **Mechanism analysis** examines the characteristics and internal mechanisms of an object to find a mathematical representation of the problem.
- The key to mechanism analysis is mathematical modeling. Solving problems by mechanism analysis is a **top-down** method.

## 1.1.1 Factstone Benchmark

- **Source: Waterloo local 2005.09.24**
- **IDs for Online Judges: POJ 2661, UVA 10916**



- Amtel has announced that it will release a 128-bit computer chip by 2010, a 256-bit computer by 2020, and so on, continuing its strategy of doubling the word-size every ten years. (Amtel released a 64-bit computer in 2000, a 32-bit computer in 1990, a 16-bit computer in 1980, an 8-bit computer in 1970, and a 4-bit computer, its first, in 1960.)

- Amtel will use a new benchmark - the *Factstone* - to advertise the vastly improved capacity of its new chips. The *Factstone* rating is defined to be the largest integer  $n$  such that  $n!$  can be represented as an unsigned integer in a computer word.
- Given a year  $1960 \leq y \leq 2160$ , what will be the *Factstone* rating of Amtel's most recently released chip?



- **Input**

- There are several test cases. For each test case, there is one line of input containing  $y$ . A line containing 0 follows the last test case.

- **Output**

- For each test case, output a line giving the Factstone rating.

# Analysis

For a given year, first the number of bits for the computer in this year is calculated, then the largest integer  $n$  (the Factstone rating) that  $n!$  can be represented as an unsigned integer in a computer word is calculated.

The computer was a 4-bit computer in 1960. Amtel doubles the word-size every ten years. That is, the number of bits for the computer in year  $Y$  is  $K = 2^{\left\lfloor \frac{Y-1960}{10} \right\rfloor}$ . The largest unsigned integer for  $K$ -bit is  $2^K - 1$ . If  $n!$  is the largest unsigned integer not greater than  $2^K - 1$ , then  $n$  is the Factstone rating in year  $Y$ .

There are two calculation methods.

Method 1: Calculate  $n!$  directly. The method is slow and leads to overflow easily.

Method 2: Logarithms are used to calculate  $n!$ . Based on the following formula:

$$\log_2 n! = \log_2 n + \log_2 (n-1) + \dots + \log_2 1 \leq \log_2 (2^K - 1) < K,$$

$n$  can be calculated. Initially  $i$  is 1, repeat  $i++$  and  $\log_2 i$  is accumulated until the sum is larger than  $K$ . Then  $i-1$  is the Factstone rating.



# Bridge

- **Source: Waterloo local 2000.09.30**
- **IDs for Online Judge: POJ 2573 , ZOJ 1877 , UVA 10037**

- $n$  people wish to cross a bridge at night. A group of at most two people may cross at any time, and each group must have a flashlight. Only one flashlight is available among the  $n$  people, so some sort of shuttle arrangement must be arranged in order to return the flashlight so that more people may cross.
- Each person has a different crossing speed; the speed of a group is determined by the speed of the slower member. Your job is to determine a strategy that gets all  $n$  people across the bridge in the minimum time.

- **Input**

- The first line of input contains  $n$ , followed by  $n$  lines giving the crossing times for each of the people. There are not more than 1000 people and nobody takes more than 100 seconds to cross the bridge.

- **Output**

- The first line of output must contain the total number of seconds required for all  $n$  people to cross the bridge. The following lines give a strategy for achieving this time. Each line contains either one or two integers, indicating which person or people form the next group to cross. (Each person is indicated by the crossing time specified in the input. Although many people may have the same crossing time the ambiguity is of no consequence.) Note that the crossings alternate directions, as it is necessary to return the flashlight so that more may cross. If more than one strategy yields the minimal time, any one will do.



# Analysis

- The **strategy** that gets all  $n$  people across the bridge in the minimum time:
  - Fast people return the flashlight to help slow people.

- A group of at most two people may cross the bridge each time
  - Solve the problem by analyzing members of groups.

- First  $n$  people's crossing times are **sorted** in descending order.
- Suppose in the current sequence,
  - **$A$**  is the current fastest person's crossing time,
  - **$B$**  is the current second fastest person's crossing time,
  - **$a$**  is the current slowest person's crossing time,
  - **$b$**  is the current second slowest person's crossing time.



- There are **two methods** making the current slowest person and the current second slowest person to cross the bridge.

- Method 1: The fastest person helps the slowest person and the second slowest person to cross the bridge.
  - Step 1: The fastest person and the slowest person cross the bridge;
  - Step 2: The fastest person is back;
  - Step 3: The fastest person and the second slowest person cross the bridge;
  - Step 4: The fastest person is back.
- It takes time  $2*A+a+b$ .

- Method 2: The fastest person and the second fastest person help the current slowest person and the current second slowest person to cross the bridge.
  - Step 1: The fastest person and the second fastest person cross the bridge;
  - Step 2: The fastest person is back and returns the flashlight to the slowest person and the second slowest person;
  - Step 3: The slowest person and the second slowest person cross the bridge and give the flashlight to the second fastest person;
  - Step 4: The second fastest person is back.
- It takes time  $2*B+A+a$ .



- Each time we need **compare** method 1 and method 2:
  - If  $(2*A+a+b < 2*B + A+a)$ , then we use method 1, else we use method 2.
- Each time the current slowest person and the current second slowest person cross the bridge.

- Finally, there are two cases:
  - **Case 1**: If there are only **two persons** need cross the bridge, then the two persons cross the bridge. It takes time  **$B$** .
  - **Case 2**: There are **three persons** need cross the bridge.
    - the fastest person and the slowest person cross the bridge;
    - the fastest person is back;
    - the last two persons cross the bridge.
- It takes time  $a + A + b$ .

## 1.2 Solving Problems by Statistical Analysis

- Statistical analysis begins with **a partial solution to the problem** and the overall global solution is found based on analyzing the partial solution.
- Solving problems by statistical analysis is a **bottom-up** method.



## 1.2.1 Ants

- **Source: Waterloo local 2004.09.19**
- **IDs for Online judges: POJ 1852, ZOJ 2376, UVA 10714**

- An army of ants walk on a horizontal pole of length  $l$  cm, each with a constant speed of 1 cm/s. When a walking ant reaches an end of the pole, it immediately falls off it. When two ants meet they turn back and start walking in opposite directions. We know the original positions of ants on the pole, unfortunately, we do not know the directions in which the ants are walking. Your task is to compute the earliest and the latest possible times needed for all ants to fall off the pole.

- **Input**

- The first line of input contains one integer giving the number of cases that follow. The data for each case start with two integer numbers: the length of the pole (in cm) and  $n$ , the number of ants residing on the pole. These two numbers are followed by  $n$  integers giving the position of each ant on the pole as the distance measured from the left end of the pole, in no particular order. All input integers are not bigger than 1000000 and they are separated by whitespace.

- **Output**

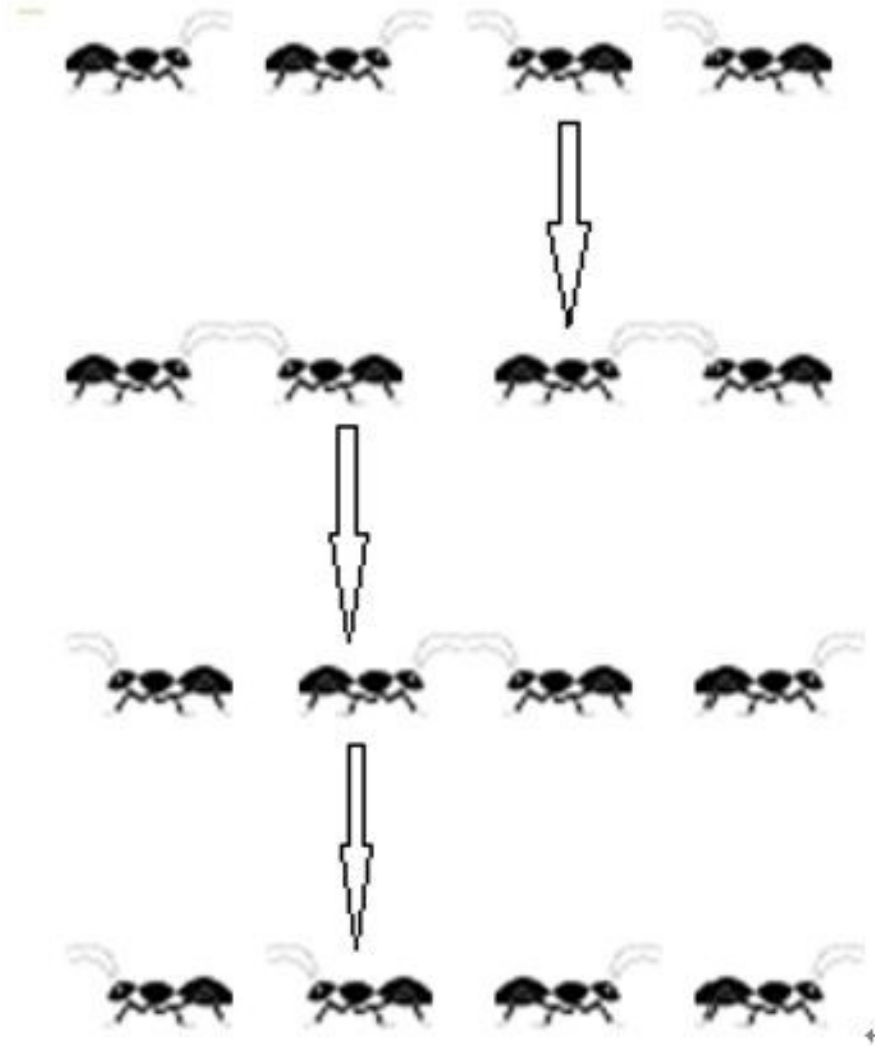
- For each case of input, output two numbers separated by a single space. The first number is the earliest possible time when all ants fall off the pole (if the directions of their walks are chosen appropriately) and the second number is the latest possible such time.





# Analysis

- The upper limit of the number of ants is 1000000.
- The upper limit of the number of combinations for ants' walking is  $2^{1000000}$ .
- The problem can't be solved by enumerating ants' walking.

First, we analyze the case that a few ants walk on a horizontal pole.



When two ants meet, that is, “”; they’ll turn back and start walking in opposite directions, that is, “”. All ants are same. Therefore, we can think all ants walk in their original directions no matter whether they meet or not. There are two values for the time that an ant falls off the pole: the ant walks to the left, or the ant walks to the right.



Suppose  $l_i$  is the position of ant  $i$  on the pole, that is, the distance measured from the left end of the pole,  $1 \leq i \leq n$ ;  $little$  is the earliest possible time when all ants fall off the pole; and  $big$  is the latest possible time when all ants fall off the pole. Based on it, the algorithm is as follows.

$$little_i = \min\{l_i, L - l_i\}, \quad big_i = \max\{l_i, L - l_i\}, \quad 1 \leq i \leq n; \quad little = \max\{little_i \mid 1 \leq i \leq n\}, \quad big = \max\{big_i \mid 1 \leq i \leq n\}.$$

## 1.2.2 Matches Game

- **Source: POJ Monthly, readchild**
- **ID for Online Judge: POJ 2234**

- Here is a simple game. In this game, there are several piles of matches and two players. The two players play in turn. In each turn, one can choose a pile and take away an arbitrary number of matches from the pile (Of course the number of matches, which is taken away, cannot be zero and cannot be larger than the number of matches in the chosen pile). If after a player's turn, there is no match left, the player is the winner. Suppose that the two players are all very clear. Your job is to tell whether the player who plays first can win the game or not.



- **Input**

- The input consists of several lines, and in each line there is a test case. At the beginning of a line, there is an integer  $M$  ( $1 \leq M \leq 20$ ), which is the number of piles. Then comes  $M$  positive integers, which are not larger than 10000000. These  $M$  integers represent the number of matches in each pile.

- **Output**

- For each test case, output "Yes" in a single line, if the player who play first will win, otherwise output "No".

# Analysis

- The problem is a Nimm's Game problem.
- Cases for the game are analyzed as follows.
- **Case 1:**
  - There is **only one pile** of matches.
  - The player who plays first will take away all matches from the pile and win the game.

- **Case 2:** There are **two piles of matches**. Numbers of matches in the two piles are  $N_1$  and  $N_2$  respectively.
  - [1] If  $N_1 \neq N_2$ ,
    - the player who plays first will take away some matches from the larger pile to make the two piles have the same number of matches. Then by mimicking the player who plays second and taking the same number of matches that he takes, just from the opposite pile, the player who plays **first** will **win** the game.
  - [2] If  $N_1 = N_2$ ,
    - the player who plays second will take the same number of matches as the player who plays first takes, just from the opposite pile, then the player who plays **second** will **win** the game.



- **Case 3:** There are **more than two piles** of matches.
- Each natural number can be represented as a binary number. For example,  $57_{(10)} = 111001_{(2)}$ , that is,  $57_{(10)} = 2^5 + 2^4 + 2^3 + 2^0$ . A pile with 57 matches can be regarded as 4 little piles, a pile with  $2^5$  matches, a pile with  $2^4$  matches, a pile with  $2^3$  matches, and a pile with  $2^0$  matches.
- Suppose there are  $k$  piles of matches,  $k > 2$ , and numbers of matches in the  $k$  piles are  $N_1, N_2, \dots$ , and  $N_k$  respectively.  $N_i$  can be represented as a  $(s+1)$ -digit binary number, that is,  $N_i = n_{is} \dots n_{i1} n_{i0}$ ,  $n_{ij}$  is a binary digit,  $0 \leq j \leq s$ ,  $1 \leq i \leq k$ . If the digit of a binary number is less than  $s+1$ , leading zeros are added.

- The game state is **balanced** if  $n_{10} + n_{20} + \dots + n_{k0}$  is even,  $n_{11} + n_{21} + \dots + n_{k1}$  is even, ....., and  $n_{1s} + n_{2s} + \dots + n_{ks}$  is even, that is,  $n_{10} \text{ XOR } n_{20} \text{ XOR } \dots \text{ XOR } n_{k0}$  is 0,  $n_{11} \text{ XOR } n_{21} \text{ XOR } \dots \text{ XOR } n_{k1}$  is 0, ....., and  $n_{1s} \text{ XOR } n_{2s} \text{ XOR } \dots \text{ XOR } n_{ks}$  is 0; **else** the game state is **unbalanced**.

- If a player faces an **unbalanced state**, he can take away some matches from a pile to make the state become a balanced state.
- And if a player faces a **balanced state**, no matter what strategies he takes, the state will become an unbalanced state.
- The final state for the game is all binary numbers are zero. That is, the final state is balanced.



- The strategy winning the game (**Bouton's Theorem**) is follows.
  - The player who plays first will win the game if the initial state is unbalanced. And the player who plays second will win the game if the initial state is balanced.

For example, there are 4 piles of matches. There are 7, 9, 12, and 15 matches in the 4 piles respectively. 7, 9, 12, and 15 can be represented as binary numbers 0111, 1001, 1100, and 1111. It is showed in the following list.

Size of a pile↵	$2^3 = 8$ ↵	$2^2 = 4$ ↵	$2^1 = 2$ ↵	$2^0 = 1$ ↵	↵
7↵	0↵	1↵	1↵	1↵	↵
9↵	1↵	0↵	0↵	1↵	↵
12↵	1↵	1↵	0↵	0↵	↵
15↵	1↵	1↵	1↵	1↵	↵
↵	Odd↵	Odd↵	Even↵	Odd↵	↵

The initial state for the game is unbalanced. The player who plays first takes away some matches from a pile to make the state become a balanced state. There are many choices. For example, the player who plays first takes away 11 matches from a pile with 12 matches to make the state become a balanced state. It is showed in the following list.

Size of a pile	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
7	0	1	1	1
9	1	0	0	1
$12 \Rightarrow 1$	0	0	0	1
15	1	1	1	1



- The method that **the player who plays first** takes away some matches from a pile **to make the state become a balanced state** is to select a row (a pile), and to flip values of bits in odd columns in the row.
- After flipping values of bits in odd columns, the number of matches is less than the original number of matches in the row. The number of matches that the player who plays first takes away from the corresponding pile is the difference between the original number of matches and the new number of matches.
- Then, **the player who plays second** takes away matches under a balanced state. The state will **become an unbalanced state**. And the player who plays first can make the state balance no matter how the player who plays second takes away matches. The process is repeated until the player who plays second takes away some matches under a balanced state last time, and then the player who plays first can take away all remainder matches.

- By the same reason, the player who plays second will win the game when the initial state is a balanced game.

- The algorithm

- $N$  piles of matches are represented as  $N$  binary numbers.
- If the initial state is unbalanced, the player who plays first will win the game, else the player who plays second will win the game.



# Greedy Algorithms

- Greedy algorithms are used to solve optimization problems through a sequence of steps.
- At each step greedy algorithms make the locally optimal choice in order to find a globally optimal solution.
- For some problems, greedy algorithms can yield a globally optimal solution; but for some problems, such as the traveling salesman problem (TSP), they can't.

# Practice for Greedy Algorithms

- Practices for Greedy Algorithms
- Greedy-Choices Based on Sorted Data
- Greedy Algorithms Used with Other Methods to Solve P-Problems

# Practices for Greedy Algorithms

- Greedy algorithms are used to solve optimization problems through a sequence of steps, and make the choice that looks best at each step.
- There are some famous greedy algorithms, such as **Prim's algorithm** and **Kruskal's algorithm** used to find a minimum spanning tree for a weighted undirected graph; **Dijkstra's algorithm** used to get single-source shortest paths between nodes in a graph; and **Huffman coding**.

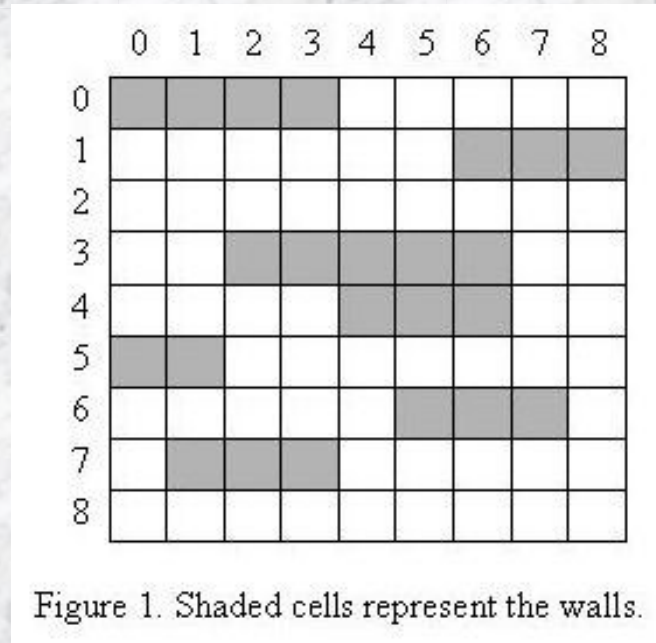


# Pass-Muraille

- **Source: ACM Tehran 2002 Preliminary**
- **IDs for Online Judges: POJ 1230 , ZOJ 1375**

- In modern day magic shows, passing through walls is very popular in which a magician performer passes through several walls in a predesigned stage show. The wall-passer (Pass-Muraille) has a limited wall-passing energy to pass through at most  $k$  walls in each wall-passing show. The walls are placed on a grid-like area. An example is shown in Figure 1, where the land is viewed from above. All the walls have unit widths, but different lengths. You may assume that no grid cell belongs to two or more walls. A spectator chooses a column of the grid. Our wall-passer starts from the upper side of the grid and walks along the entire column, passing through every wall in his way to get to the lower side of the grid. If he faces more than  $k$  walls when he tries to walk along a column, he would fail presenting a good show.

- For example, in the wall configuration shown in Figure 1, a wall-passer with  $k = 3$  can pass from the upper side to the lower side choosing any column except column 6.





- Given a wall-passer with a given energy and a show stage, we want to remove the minimum number of walls from the stage so that our performer can pass through all the walls at any column chosen by spectators.

- **Input**

- The first line of the input file contains a single integer  $t$  ( $1 \leq t \leq 10$ ), the number of test cases, followed by the input data for each test case. The first line of each test case contains two integers  $n$  ( $1 \leq n \leq 100$ ), the number of walls, and  $k$  ( $0 \leq k \leq 100$ ), the maximum number of walls that the wall-passer can pass through, respectively. After the first line, there are  $n$  lines each containing two  $(x, y)$  pairs representing coordinates of the two endpoints of a wall. Coordinates are non-negative integers less than or equal to 100. The upper-left of the grid is assumed to have coordinates  $(0, 0)$ . The second sample test case below corresponds to the land given in Figure 1.

- **Output**

- There should be one line per test case containing an integer number which is the minimum number of walls to be removed such that the wall-passer can pass through walls starting from any column on the upper side.

# Analysis

- All columns are scanned from left to right.
- Removing the minimum number of walls from the stage must guarantee removing the minimum number of walls in scanned columns.
- The optimal solution to the problem consists of its optimal solutions to subproblems. The key to the problem is its greedy-choice.



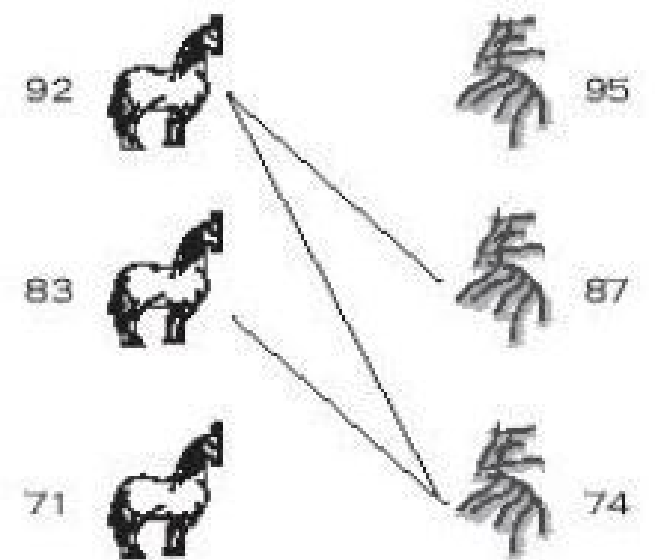
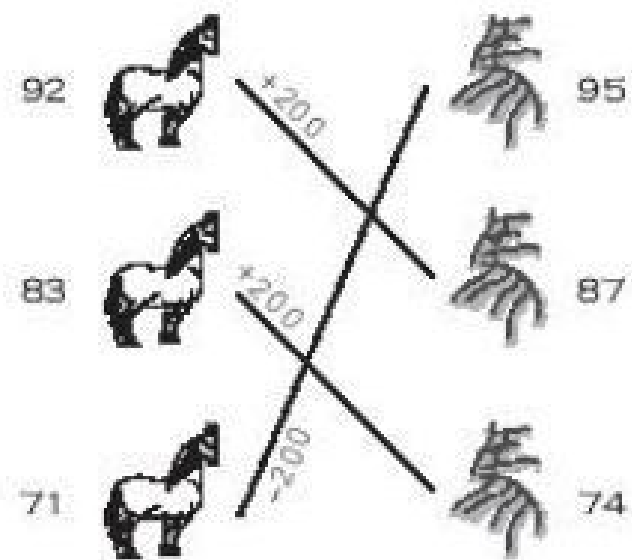
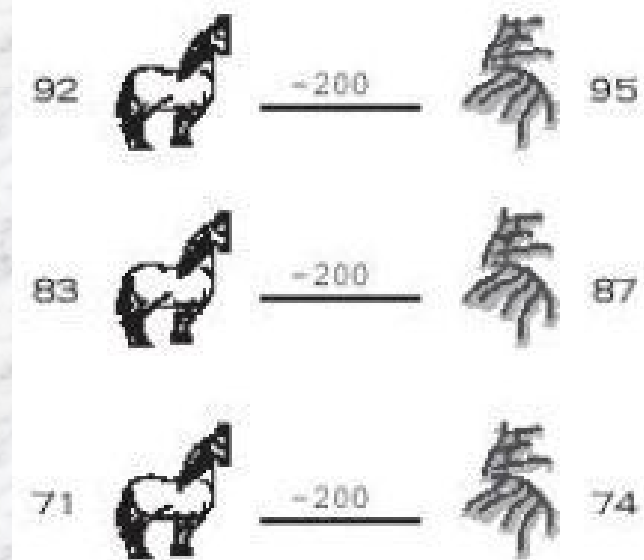
- Suppose there are  $D$  walls in the current column.
- If  $D \leq K$ , we needn't remove any wall; and if  $D > K$ ,  $D-K$  walls must be removed.
- **The greedy-choice**
  - For walls in the current column, the longest  $D-K$  walls in unscanned columns are removed.
  - The greedy-choice removes minimum number of walls.

## 5.1.2 Tian Ji-- The Horse Racing

- **Source: ACM Shanghai 2004**
- **IDs for Online Judges: POJ 2287 , ZOJ 2397 , UVA 3266**

- Here is a famous story in Chinese history.
- That was about 2300 years ago. General Tian Ji was a high official in the country Qi. He likes to play horse racing with the king and others.
- Both of Tian and the king have three horses in different classes, namely, regular, plus, and super. The rule is to have three rounds in a match; each of the horses must be used in one round. The winner of a single round takes two hundred silver dollars from the loser.
- Being the most powerful man in the country, the king has so nice horses that in each class his horse is better than Tian's. As a result, each time the king takes six hundred silver dollars from Tian.
- Tian Ji was not happy about that, until he met Sun Bin, one of the most famous generals in Chinese history. Using a little trick due to Sun, Tian Ji brought home two hundred silver dollars and such a grace in the next match.
- It was a rather simple trick. Using his regular class horse race against the super class from the king, they will certainly lose that round. But then his plus beat the king's regular, and his super beat the king's plus. What a simple trick. And how do you think of Tian Ji, the high ranked official in China?





- Were Tian Ji lives in nowadays, he will certainly laugh at himself. Even more, were he sitting in the ACM contest right now, he may discover that the horse racing problem can be simply viewed as finding the maximum matching in a bipartite graph. Draw Tian's horses on one side, and the king's horses on the other. Whenever one of Tian's horses can beat one from the king, we draw an edge between them, meaning we wish to establish this pair. Then, the problem of winning as many rounds as possible is just to find the maximum matching in this graph. If there are ties, the problem becomes more complicated, he needs to assign weights 0, 1, or -1 to all the possible edges, and find a maximum weighted perfect matching...
- However, the horse racing problem is a very special case of bipartite matching. The graph is decided by the speed of the horses -- a vertex of higher speed always beat a vertex of lower speed. In this case, the weighted bipartite matching algorithm is a too advanced tool to deal with the problem.
- In this problem, you are asked to write a program to solve this special case of matching problem.

- **Input**

- The input consists of up to 50 test cases. Each case starts with a positive integer  $n$  ( $n \leq 1000$ ) on the first line, which is the number of horses on each side. The next  $n$  integers on the second line are the speeds of Tian's horses. Then the next  $n$  integers on the third line are the speeds of the king's horses. The input ends with a line that has a single '0' after the last test case.

- **Output**

- For each input case, output a line containing a single number, which is the maximum money Tian Ji will get, in silver dollars.



# Analysis

- The problem can be solved by several different methods. Maximum matching in a bipartite graph or dynamic programming can be used to solve the problem, but using greedy algorithm to solve the problem is simple and efficient. The greedy algorithm is as follow.
- First, the speeds of Tian's horses and the speeds of the king's horses are sorted in ascending order respectively. Suppose the sequence for speeds of Tian's current horses in ascending order is  $A=a_1...a_n$ ; and the sequence for the speeds of the king's current horses are sorted in ascending order is  $B=b_1...b_n$ .

- Second, greedy-choices are as follow.
- 1. If Tian's current slowest horse is faster than the king's current slowest horse, that is,  $a_1 > b_1$ ; then Tian's current slowest horse races against the king's current slowest horse, that is,  $a_1$  is compared with  $b_1$ . Because  $b_1$  is less than any elements in  $A$  and the king's current slowest horse can be defeated by any Tian's remainder horse, it is suitable that the king's current slowest horse is defeated by Tian's current slowest horse.
- 2. If Tian's current slowest horse is slower than the king's current slowest horse, that is,  $a_1 < b_1$ ; then Tian's current slowest horse races against the king's current fastest horse, that is,  $a_1$  is compared with  $b_n$ . Because  $a_1$  is less than any elements in  $B$  and Tian's current slowest horse can be defeated by any king's remainder horse, it is suitable that Tian's current slowest horse is defeated by the king's current fastest horse.



- 3. If Tian's current fastest horse is faster than the king's current fastest horse, that is,  $a_n > b_n$ ; then Tian's current fastest horse races against the king's current fastest horse, that is,  $a_n$  is compared with  $b_n$ . Because  $a_n$  is larger than any elements in  $B$  and Tian's current fastest horse can defeat any king's remainder horse, it is suitable that Tian's current fastest horse defeats the king's current fastest horse.
- 4. If Tian's current fastest horse is slower than the king's current fastest horse, that is,  $a_n < b_n$ ; then Tian's current slowest horse races against the king's current fastest horse, that is,  $a_1$  is compared with  $b_n$ . Because  $b_n$  is larger than any elements in  $A$  and the king's current fastest horse can defeat any Tian's remainder horse, it is suitable that the king's current fastest horse defeats Tian's current slowest horse.



- 5. If  $(a_1 == b_1)$  and  $(a_n > b_n)$ , then it is suitable that Tian's current fastest horse races against the king's current fastest horse, that is,  $a_n$  is compared with  $b_n$ .
- 6. If  $(a_n == b_n)$ , then there exist an optimal solution that  $a_1$  is compared with  $b_n$ .
- The above process repeats until the horse racing ends. Tian's current fastest or slowest horse races against the king's current fastest or slowest horse each time based on above greedy-choices. Optimal solutions to subproblems constitute the global optimal solution to the problem.

# Greedy-Choices based on Sorted Data

- The key to a greedy algorithm is its greedy-choices.
- Sometimes the greedy-choices must be based on sorted data:
  - First, data are sorted.
  - Then, greedy-choices are made based on sorted data.

# **Radar Installation**

- **Source: ACM Beijing 2002**
- **IDs for Online Judge: POJ 1328 , ZOJ 1360 , UVA 2519**



- Assume the coasting is an infinite straight line. Land is in one side of coasting, sea in the other. Each small island is a point locating in the sea side. And any radar installation, locating on the coasting, can only cover  $d$  distance, so an island in the sea can be covered by a radius installation, if the distance between them is at most  $d$ .
- We use Cartesian coordinate system, defining the coasting is the x-axis. The sea side is above x-axis, and the land side below. Given the position of each island in the sea, and given the distance of the coverage of the radar installation, your task is to write a program to find the minimal number of radar installations to cover all the islands. Note that the position of an island is represented by its x-y coordinates.

- **Input**

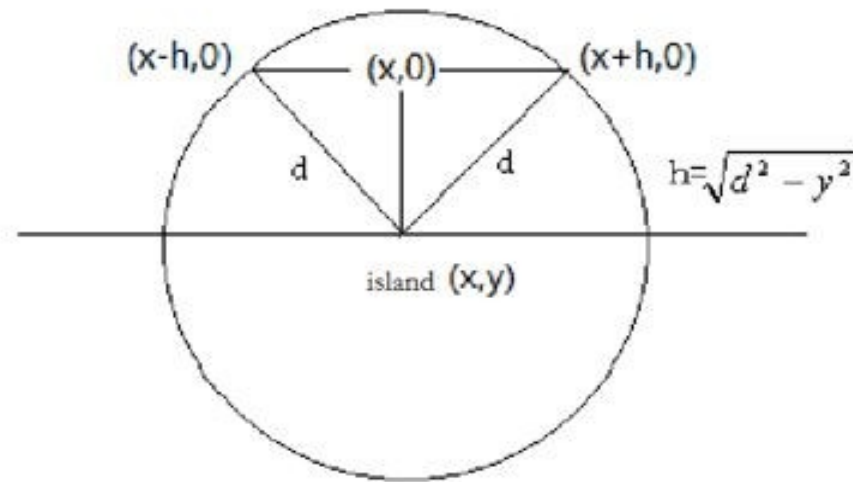
- The input consists of several test cases. The first line of each case contains two integers  $n$  ( $1 \leq n \leq 1000$ ) and  $d$ , where  $n$  is the number of islands in the sea and  $d$  is the distance of coverage of the radar installation. This is followed by  $n$  lines each containing two integers representing the coordinate of the position of each island. Then a blank line follows to separate the cases. The input is terminated by a line containing pair of zeros.

- **Output**

- For each test case output one line consisting of the test case number followed by the minimal number of radar installations needed. "-1" installation means no solution for that case.

# Analysis

Each small island is represented as a segment on the coasting. If a rader locates on the segment, the island can be covered by the radar. Suppose the Cartesian coordinate for the island is  $(x, y)$ . If a rader locates on the coasting from  $(x-h, 0)$  to  $(x+h, 0)$ , where  $h = \sqrt{d^2 - y^2}$ , the island can be covered. Therefor the island is represented as a segment from  $(x-h, 0)$  to  $(x+h, 0)$ . It can be showed as Figure.





- Suppose there are  $n$  islands.
- First  $n$  islands are represented as  $n$  segments.
- Second, right endpoints are as the first key (in ascending order), left endpoints are as second key (in ascending order), and the  $n$  segments are sorted.
- Finally all sorted segments are scanned one by one. If the current segment isn't covered by a radar, a radar locates at the right endpoint for the segment.

# Practice for Dynamic Programming

- **Dynamic Programming (DP)** is used to solve optimization problems.
- **Dynamic programming** breaks an optimization problem into a sequence of related subproblems, solves these subproblems just once, stores solutions to subproblems, and constructs an optimal solution to the problem based on solutions to subproblems.
- The method storing solutions to subproblems is called **memorization**. When the same subproblem occurs, its solution can be used directly.

- There are **two characters** for a problem solved by Dynamic Programming.
- [1] **Optimization**. An optimal solution to a problem consists of optimal solutions to subproblems.
- [2] **No aftereffect**. A solution to a subproblem is only related to solutions to its direct predecessors.



# Practice for Dynamic Programming

- Linear Dynamic Programming
  - Linear Dynamic Programming
  - Subset Sum
  - Longest Common Subsequence (LCS)
  - Longest Increasing Subsequence (LIS)
- Tree-Like Dynamic Programming
- Dynamic Programming with State Compression

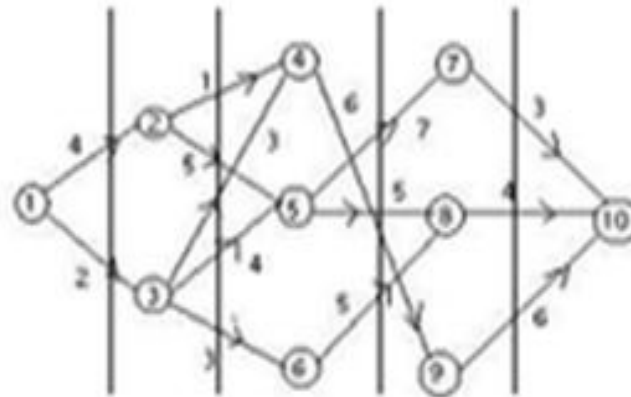
# Linear Dynamic Programming

- Linear Dynamic Programming
- Subset Sum
- Longest Common Subsequence (LCS)
- Longest Increasing Subsequence (LIS)

# Linear Dynamic Programming

**Stage  $k$  and State  $s_k$ :** The solution to a problem is divided into  $k$  orderly and related stages. In a stage there are several states. State  $s_k$  is a state in stage  $k$ .

For example, the following figure shows a solution to a problem is divided into 5 orderly and related stages. State 1 is called the initial state, in stage 1. State 10 is called the goal state, in stage 5. In stage 3 there are 3 states: state 4, state 5, and state 6.





**Decision  $u_k$  and Available Decision Set  $D_k(s_k)$ :** The choice from a state in stage  $k-1$  (the current stage) to a state in stage  $k$  (the next stage) is called decision  $u_k$ . Normally a state can be reachable through more than one decision from the last stage, and such decisions constitute an available decision set  $D_k(s_k)$ .

For example, there are two decisions reaching state 5:  $2 \rightarrow 5$ ,  $3 \rightarrow 5$ ,  $D_3(5) = \{2, 3\}$ . A decision sequence from the initial state to the goal state is called a strategy. For example,  $1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 10$  is a strategy.

**Successor Function and Optimization:** A successor function is used to describe the transition from stage  $k-1$  to stage  $k$ . Dynamic programming method is used to solve some optimization problems. Successor functions are used to find a solution with the optimal (minimum or maximum) value to a problem. A successor function can be formally defined as follow.

$$f_k(s_k) = \text{opt } g(f_{k-1}(T_k(s_k, u_k)), u_k);$$

where  $T_k(s_k, u_k)$  is a state  $s_{k-1}$  in stage  $k-1$  which relate to state  $s_k$  through decision  $u_k$ , and  $f_{k-1}(T_k(s_k, u_k))$  is an optimal solution,  $g(x, u_k)$  is a function for value  $x$  and decision  $u_k$ , that is,  $g(f_{k-1}(T_k(s_k, u_k)))$  is a function from state  $s_{k-1}$  to state  $s_k$  through decision  $u_k$ ; *opt* means optimization;  $f_1(s_1)$  is an initial value. Because  $u_k$  is one decision in decision set  $D_k(s_k)$ , all decisions are enumerated to get the optimal solution to  $s_k$ . From initial state, successor functions are used to get the optimal solution  $f_n$  (goal state) to the problem finally.

for ( every stage  $i$  is processed in linear order)  $\leftarrow$

{  $\leftarrow$

for ( every state  $j$  in stage  $i$  is enumerated ( $j \in S_i$ ) )  $\leftarrow$

{ for ( every state  $k$  in stage  $i-1$  which is related to state  $j$  is enumerated ( $k \in S_{i-1}$ ) )  $\leftarrow$

{ calculate  $f_i(j) = \underset{u_k \in D_k(k)}{opt} g(f_{i-1}(k), u_k);$  }  $\leftarrow$

}  $\leftarrow$

}  $\leftarrow$



# Brackets Sequence

- **Source: ACM Northeastern Europe 2001**
- **IDs for Online Judges: POJ 1141 , ZOJ 1463 , Ural 1183 , UVA 2451**

- Let us define a regular brackets sequence in the following way:
- 1. Empty sequence is a regular sequence.
- 2. If  $S$  is a regular sequence, then  $(S)$  and  $[S]$  are both regular sequences.
- 3. If  $A$  and  $B$  are regular sequences, then  $AB$  is a regular sequence.
- For example, all of the following sequences of characters are regular brackets sequences:
  - $()$ ,  $[]$ ,  $((()))$ ,  $([])$ ,  $()[]$ ,  $()[()]$
- And all of the following character sequences are not:
  - $($ ,  $[$ ,  $)$ ,  $)$ (,  $([])$ ,  $([($
- Some sequence of characters  $('(', ')')$ ,  $('['$ , and  $']')$  is given. You are to find the shortest possible regular brackets sequence, that contains the given character sequence as a subsequence. Here, a string  $a_1 a_2 \dots a_n$  is called a subsequence of the string  $b_1 b_2 \dots b_m$ , if there exist such indices  $1 \leq i_1 < i_2 < \dots < i_n \leq m$ , that  $a_j = b_{i_j}$  for all  $1 \leq j \leq n$ .

- **Input**

- The input file contains at most 100 brackets (characters '(', ')', '[' and ']') that are situated on a single line without any other characters among them.

- **Output**

- Write to the output file a single line that contains some regular brackets sequence that has the minimal possible length and contains the given sequence as a subsequence.



# Analysis

Suppose stage  $r$  is the length of subsequence,  $1 \leq r \leq n$ ; and state  $i$  is the pointer pointing to the front of the current subsequence,  $0 \leq i \leq n-r$ . Based on  $i$  and  $r$ , the pointer  $j$  pointing to the rear of the current subsequence can be calculated,  $j=i+r-1$ . Suppose  $dp[i, j]$  is the minimal number of characters that must be inserted into  $s_i \dots s_j$ . Obviously, if the length of subsequence is 1,  $dp[i, i]=1$ ,  $0 \leq i < \text{strlen}(s)$ . ↵

If  $((s_i == '[') \ \&\& \ (s_j == ']')) \parallel ((s_i == '(') \ \&\& \ (s_j == ')'))$ , then the minimal number of characters that must be inserted into  $s_i \dots s_j$  is the minimal number of characters that must be inserted into  $s_{i+1} \dots s_{j-1}$ , that is,  $dp[i, j] = dp[i+1, j-1]$ ; otherwise  $s_i \dots s_j$  is divided into two parts, and we need to determine the pointer  $k$  ( $i \leq k < j$ ) so that  $dp[i, j] = \min_{1 \leq k < j} (dp[i, k] + dp[k+1, j])$ . ↵

Based on it, a memorized list path[ ][ ] is used to store all solutions to subproblems:↵

$$\text{path}[i][j] = \begin{cases} -1 & ((s_i == '[') \& \& (s_j == ']')) \parallel ((s_i == '(') \& \& (s_j == ')')) \\ k & dp[i, j] = \min_{i \leq k < j} (dp[i, k] + dp[k + 1, j]) \end{cases} \quad \cdot \text{↵}$$

After memorized list path[ ][ ] is calculated through DP, regular brackets sequence that has the minimal possible length and contains the given sequence as a subsequence can be gotten through recursion.↵

# Subset Sum

- Suppose  $S = \{ x_1, x_2, \dots, x_n \}$  is a set of non-negative integers, and  $c$  is a non-negative integer.
- Subset Sum Problem is to determine whether there is a subset of the given set with sum equal to given  $c$ .



Coin Counting is a classical problem for subset sum. Given a set of  $n$  non-negative integers  $\{a_1, a_2, \dots, a_n\}$  and a non-negative integer  $T$ , Coin Counting is to determine how many solutions to  $k_1a_1 + k_2a_2 + \dots + k_na_n = T$ , where  $k_1, k_2, \dots, k_n$  are non-negative integers. DP can be used to solve the problem. Suppose  $c(i, j)$  is the number of solutions to  $k_1a_1 + k_2a_2 + \dots + k_ia_i = j$ ,  $k_i \geq 0$ . Obviously the goal for Coin Counting is to calculate  $c(n, T)$ . In order to calculate  $c(i, j)$ , stage  $i$  is the first  $i$  integers are used,  $1 \leq i \leq n$ ; states are  $k_1a_1 + k_2a_2 + \dots + k_ia_i = j$ ,  $0 \leq j \leq T$ . The successor function is as follow.

$$c(i, j) = \begin{cases} 1 & i = 0 \\ \sum_{k=1}^{i-1} c(k, j - a_i) & i \geq 1, j \geq a_i \end{cases}$$

The final solution is  $c(n, T)$ .

# Dollars

- **Source: New Zealand Contest 1991**
- **IDs for Online Judge: UVA 147**

- New Zealand currency consists of \$100, \$50, \$20, \$10, and \$5 notes and \$2, \$1, 50c, 20c, 10c and 5c coins. Write a program that will determine, for any given amount, in how many ways that amount may be made up. Changing the order of listing does not increase the count. Thus 20c may be made up in 4 ways:  $1 \times 20c$ ,  $2 \times 10c$ ,  $10c + 2 \times 5c$ , and  $4 \times 5c$ .



- **Input**

- Input will consist of a series of real numbers no greater than \$50.00 each on a separate line. Each amount will be valid, that is will be a multiple of 5c. The file will be terminated by a line containing zero (0.00).

- **Output**

- Output will consist of a line for each of the amounts in the input, each line consisting of the amount of money (with two decimal places and right justified in a field of width 5), followed by the number of ways in which that amount may be made up, right justified in a field of width 12.

# Analysis

- First, DP is used to calculate all solutions to the problem in the range. 5c coin is the smallest coin. Other notes and coins for New Zealand currency are multiples for 5c coin. Therefore 5c coin is used as the unit for notes and coins for New Zealand currency. Suppose  $b[i]$  is the number of 5c coins for the  $i$ -th currency,  $0 \leq i \leq 10$ ;  $a[i, j]$  is the number of ways in which  $j$  5c coins may be made up using the first  $i$ -th currencies,  $0 \leq i \leq 10$ ,  $0 \leq j \leq 6000$ .
- Obviously, the number of way in which  $j$  5c coins may be made up only using 5c coin is 1, that is,  $a[0, j] = 1$ ,  $0 \leq j \leq 6000$ . If the amount is equal to a coin or a note, there is a way that the amount may be made up using the coin or the note.



- For 10 cents, there are two ways. 10 cents are made up using 5c coins or 10c coin.
- For 15 cents, the first way is 15c cents are made up only using 5c coins. Then we calculate the number of ways in which 15 cents are made up using 5c coin and 10c coin (the way only using 5c coin needn't to be considered). First a 10c coin is used (At least one 10c coin is used), and then a 5c coin is used. Therefore there are two ways for 15 cents.
- For 20 cents, the first case is only 5c coins are used. For the second case is a 10 coin is used firstly (At least one 10c coin is used), and for remainder 10 cents there are two ways. The final case is only 20c coin is used. Therefore there are four ways.



Based on it, the number of ways in which  $j$  5c coins may be made up using the first  $i$ -th currencies is based on the number of ways in which  $j-b[i]$  5c coins may be made up using the first  $(i-1)$ -th currencies. That is,  $a[i, j] = \sum_{k=0}^{i-1} a[k, j-b[i]] \mid j \geq b[i]$ .

Then, for each test case, the solution can be gotten based on array  $a$ . For a real number  $n$ , the solution is  $a[10, \lfloor n*20 \rfloor]$ .

The problem can also be solved by generation function.



