Greedy Algorithm

Greedy Algorithm

Sample Problem: Barn Repair [1999 USACO Spring Open]

There is a long list of stalls, some of which need to be covered with boards. You can use up to N (1 <= N <= 50) boards, each of which may cover any number of consecutive stalls. Cover all the necessary stalls, while covering as few total stalls as possible.

The Idea

The basic idea behind greedy algorithms is to build large solutions up from smaller ones. Unlike other approaches, however, greedy algorithms keep only the best solution they find as they go along. Thus, for the sample problem, to build the answer for N = 5, they find the best solution for N = 4, and then alter it to get a solution for N = 5. No other solution for N = 4 is ever considered.

Greedy algorithms are **fast**, generally linear to quadratic and require little extra memory. Unfortunately, they usually aren't correct. But when they do work, they are often easy to implement and fast enough to execute.

Problems

There are two basic problems to greedy algorithms.

How to Build

How does one create larger solutions from smaller ones? In general, this is a function of the problem. For the sample problem, the most obvious way to go from four boards to five boards is to pick a board and remove a section, thus creating two boards from one. You should choose to remove the largest section from any board which covers only stalls which don't need covering (so as to minimize the total number of stalls covered).

To remove a section of covered stalls, take the board which spans those stalls, and make into two boards: one of which covers the stalls before the section, one of which covers the stalls after the section.

Does it work?

The real challenge for the programmer lies in the fact that greedy solutions don't always work. Even if they seem to work for the sample input, random input, and all the cases you can think of, if there's a case where it won't work, at least one (if not more!) of the judges' test cases will be of that form.

For the sample problem, to see that the greedy algorithm described above works, consider the following:

Assume that the answer doesn't contain the large gap which the algorithm removed, but does contain a gap which is smaller. By combining the two boards at the end of the smaller gap and splitting the board across the larger gap, an answer is obtained which uses as many boards as the original solution but which covers fewer stalls. This new answer is better, so therefore the assumption is wrong and we should always choose to remove the largest gap.

If the answer doesn't contain this particular gap but does contain another gap which is just as large, doing the same transformation yields an answer which uses as many boards and covers as many stalls as the other answer. This new answer is just as good as the original solution but no better, so we may choose either.

Thus, there exists an optimal answer which contains the large gap, so at each step, there is always an optimal answer which is a superset of the current state. Thus, the final answer is optimal.

Conclusions

If a greedy solution exists, use it. They are easy to code, easy to debug, run quickly, and use little memory, basically defining a good algorithm in contest terms. The only missing element from that list is correctness. If the greedy algorithm finds the correct answer, go for it, but don't get suckered into thinking the greedy solution will work for all problems.

Sample Problems

Sorting a three-valued sequence [IOI 1996]

You are given a three-valued (1, 2, or 3) sequence of length up to 1000. Find a minimum set of exchanges to put the sequence in sorted order.

Algorithm The sequence has three parts: the part which will be 1 when in sorted order, 2 when in sorted order, and 3 when in sorted order. The greedy algorithm swaps as many as possible of the 1's in the 2 part with 2's in the 1 part, as many as possible 1's in the 3 part with 3's in the 1 part, and 2's in the 3 part with 3's in the 2 part. Once none of these types remains, the remaining elements out of place need to be rotated one way or the other in sets of 3. You can optimally sort these by swapping all the 1's into place and then all the 2's into place.

Analysis: Obviously, a swap can put at most two elements in place, so all the swaps of the first type are optimal. Also, it is clear that they use different types of elements, so there is no ``interference'' between those types. This means the order does not matter. Once those swaps have been performed, the best you can do is two swaps for every three elements not in the correct location, which is what the second part will achieve (for example, all the 1's are put in place but no others; then all that remains are 2's in the 3's place and vice-versa, and which can be swapped).

Friendly Coins - A Counterexample [abridged]

Given the denominations of coins for a newly founded country, the Dairy Republic, and some monetary amount, find the smallest set of coins that sums to that amount. The Dairy Republic is guaranteed to have a 1 cent coin.

Algorithm: Take the largest coin value that isn't more than the goal and iterate on the total minus this value.

(Faulty) Analysis: Obviously, you'd never want to take a smaller coin value, as that would mean you'd have to take more coins to make up the difference, so this algorithm works.

Maybe not: Okay, the algorithm usually works. In fact, for the U.S. coin system {1, 5, 10, 25}, it always yields the optimal set. However, for other sets, like {1, 5, 8, 10} and a goal of 13, this greedy algorithm would take one 10, and then three 1's, for a total of four coins, when the two coin solution {5, 8} also exists.

Topological Sort

Given a collection of objects, along with some ordering constraints, such as "A must be before B," find an order of the objects such that all the ordering constraints hold.

Algorithm: Create a directed graph over the objects, where there is an arc from A to B if "A must be before B." Make a pass through the objects in arbitrary order. Each time you find an object with in-degree of 0, greedily place it on the end of the current ordering, delete all of its out-arcs, and recurse on its (former) children, performing the same check. If this algorithm gets through all the objects without putting every object in the ordering, there is no ordering which satisfies the constraints.

Mixing Milk

The Merry Milk Makers company buys milk from farmers, packages it into attractive 1- and 2-Unit bottles, and then sells that milk to grocery stores so we can each start our day with delicious cereal and milk.

Since milk packaging is such a difficult business in which to make money, it is important to keep the costs as low as possible. Help Merry Milk Makers purchase the farmers' milk in the cheapest possible manner. The MMM company has an extraordinarily talented marketing department and knows precisely how much milk they need each day to package for their customers.

The company has contracts with several farmers from whom they may purchase milk, and each farmer has a (potentially) different price at which they sell milk to the packing plant. Of course, a herd of cows can only produce so much milk each day, so the farmers already know how much milk they will have available.

Each day, Merry Milk Makers can purchase an integer number of units of milk from each farmer, a number that is always less than or equal to the farmer's limit (and might be the entire production from that farmer, none of the production, or any integer in between).

Given:

- The Merry Milk Makers' daily requirement of milk
- The cost per unit for milk from each farmer
- The amount of milk available from each farmer

calculate the minimum amount of money that Merry Milk Makers must spend to meet their daily need for milk.

Note: The total milk produced per day by the farmers will always be sufficient to meet the demands of the Merry Milk Makers even if the prices are high.

PROGRAM NAME: milk

INPUT FORMAT

Line 1:	Two integers, N and M. The first value, N, (0 <= N <= 2,000,000) is the amount of milk that Merry Milk Makers wants per day. The second, M, (0 <= M <= 5,000) is the number of farmers that they may buy from.
Lines 2 through M+1:	The next M lines each contain two integers: P_i and A_i. P_i (0 <= P_i <= 1,000) is price in cents that farmer i charges. A_i (0 <= Ai <= 2,000,000) is the amount of milk that farmer i can sell to Merry Milk Makers per day.

SAMPLE INPUT (file milk.in)

```
100 5
5 20
9 40
3 10
```

```
8 80
6 30
```

INPUT EXPLANATION

`100 5` -- MMM wants 100 units of milk from 5 farmers
`5 20` -- Farmer 1 says, "I can sell you 20 units at 5 cents per unit"
`9 40` etc.
`3 10` -- Farmer 3 says, "I can sell you 10 units at 3 cents per unit"
`8 80` etc.
`6 30` -- Farmer 5 says, "I can sell you 30 units at 6 cents per unit"

OUTPUT FORMAT

A single line with a single integer that is the minimum cost that Merry Milk Makers msut pay for one day's milk.

SAMPLE OUTPUT (file milk.out)

```
630
```

OUTPUT EXPLANATION

Here's how the MMM company spent only 630 cents to purchase 100 units of milk:

Price per unit	Units available	Units bought	Price * # units	Total cost	Notes
5	20	20	5*20	100	
9	40	0			Bought no milk from farmer 2
3	10	10	3*10	30	
8	80	40	8*40	320	Did not buy all 80 units!
6	30	30	6*30	180	
Total	180	**100**		**630**	Cheapest total cost

Submission file Name: 파일 선택 선택한 파일 없음 Send it in!

USACO Gateway | Comment or Question

USA COMPUTING OLYMPIAD

Barn Repair

It was a dark and stormy night that ripped the roof and gates off the stalls that hold Farmer John's cows. Happily, many of the cows were on vacation, so the barn was not completely full.

The cows spend the night in stalls that are arranged adjacent to each other in a long line. Some stalls have cows in them; some do not. All stalls are the same width.

Farmer John must quickly erect new boards in front of the stalls, since the doors were lost. His new lumber supplier will supply him boards of any length he wishes, but the supplier can only deliver a small number of total boards. Farmer John wishes to minimize the total length of the boards he must purchase.

Given M (1 <= M <= 50), the maximum number of boards that can be purchased; S (1 <= S <= 200), the total number of stalls; C (1 <= C <= S) the number of cows in the stalls, and the C occupied stall numbers (1 <= stall_number <= S), calculate the minimum number of stalls that must be blocked in order to block all the stalls that have cows in them.

Print your answer as the total number of stalls blocked.

PROGRAM NAME: barn1

INPUT FORMAT

Line 1:	M, S, and C (space separated)
Lines 2-C+1:	Each line contains one integer, the number of an occupied stall.

SAMPLE INPUT (file barn1.in)

```
4 50 18
3
4
6
8
14
15
16
17
21
25
26
27
30
31
40
41
42
43
```

OUTPUT FORMAT

A single line with one integer that represents the total number of stalls blocked.

SAMPLE OUTPUT (file barn1.out)

```
25
```

[One minimum arrangement is one board covering stalls 3-8, one covering 14-21, one covering 25-31, and one covering 40-43.]

Submission file Name: 파일 선택 선택한 파일 없음 Send it in!

Crafting Winning Solutions

A good way to get a competitive edge is to write down a game plan for what you're going to do in a contest round. This will help you script out your actions, in terms of what to do both when things go right and when things go wrong. This way you can spend your thinking time in the round figuring out programming problems and not trying to figure out what the heck you should do next... it's sort of like precomputing your reactions to most situations.

Mental preparation is also important.

Game Plan For A Contest Round

Read through ALL the problems FIRST; sketch notes with algorithm, complexity, the numbers, data structs, tricky details, ...

- Brainstorm many possible algorithms - then pick the stupidest that works!
- DO THE MATH! (space & time complexity, and plug in actual expected and worst case numbers)
- Try to break the algorithm - use special (degenerate?) test cases
- Order the problems: shortest job first, in terms of your effort (shortest to longest: done it before, easy, unfamiliar, hard)

Coding a problem - For each, one at a time:

- Finalize algorithm
- Create test data for tricky cases
- Write data structures
- Code the input routine and test it (write extra output routines to show data?)
- Code the output routine and test it
- Stepwise refinement: write comments outlining the program logic
- Fill in code and debug *one section at a time*
- Get it working & verify correctness (use trivial test cases)
- Try to break the code - use special cases for code correctness
- Optimize progressively - only as much as needed, and keep all versions (use hard test cases to figure out actual runtime)

Time management strategy and "damage control" scenarios

Have a plan for what to do when various (foreseeable!) things go wrong; imagine problems you might have and figure out how you want to react. The central question is: "When do you spend more time debugging a program, and when do you cut your losses and move on?". Consider these issues:

- How long have you spent debugging it already?
- What type of bug do you seem to have?
- Is your algorithm wrong?
- Do you data structures need to be changed?
- Do you have any clue about what's going wrong?
- A short amount (20 mins) of debugging is better than switching to anything else; but you might be able to solve another from scratch in 45 mins.

- When do you go back to a problem you've abandoned previously?
- When do you spend more time optimizing a program, and when do you switch?
- Consider from here out - forget prior effort, focus on the future: how can you get the most points in the next hour with what you have?

Have a checklist to use before turning in your solutions:

Code freeze five minutes before end of contest?
- Turn asserts off.
- Turn off debugging output.

Tips & Tricks

- Brute force it when you can
- KISS: Simple is smart!
- Hint: focus on *limits* (specified in problem statement)
- Waste memory when it makes your life easier (if you can get away with it)
- Don't delete your extra debugging output, comment it out
- Optimize progressively, and only as much as needed
- Keep all working versions!
- Code to debug:
    - whitespace is good,
    - use meaningful variable names,
    - don't reuse variables,
    - stepwise refinement,
    - COMMENT BEFORE CODE.
- Avoid pointers if you can
- Avoid dynamic memory like the plague: statically allocate everything.
- Try not to use floating point; if you have to, put tolerances in everywhere (never test equality)
- Comments on comments:
    - Not long prose, just brief notes
    - Explain high-level functionality: ++i; /* increase the value of i by */ is worse than useless
    - Explain code trickery
    - Delimit & document functional sections
    - As if to someone intelligent who knows the problem, but not the code
    - Anything you had to think about
    - Anything you looked at even once saying, "now what does that do again?"
    - Always comment order of array indices
- Keep a log of your performance in each contest: successes, mistakes, and what you could have done better; use this to rewrite and improve your game plan!

Complexity

Basics and order notation

The fundamental basis of complexity analysis revolves around the notion of ``big oh'' notation, for instance: O(N). This means that the algorithm's execution speed or memory usage will double when the problem size doubles. An algorithm of O(N^2) will run about four times slower (or use 4x more space) when the problem size doubles. Constant-time or space algorithms are denoted O(1). This concept applies to time and space both; here we will concentrate discussion on time.

One deduces the O() run time of a program by examining its loops. The most nested (and hence slowest) loop dominates the run time and is the only one mentioned when discussing O() notation. A program with a single loop and a nested loop (presumably loops that execute *N* times each) is O(N^2), even though there is also a O(*N*) loop present.

Of course, recursion also counts as a loop and recursive programs can have orders like O(b^N), O(*N!*), or even O(N^N).

Rules of thumb

- When analyzing an algorithm to figure out how long it might run for a given dataset, the first rule of thumb is: modern (2004) computers can deal with 100M actions per second. In a five second time limit program, about 500M actions can be handled. Really well optimized programs might be able to double or even quadruple that number. Challenging algorithms might only be able to handle half that much. Current contests usually have a time limit of 1 second for large datasets.
- 16MB maximum memory use
- 2^{10} ~approx~ 10^3
- If you have *k* nested loops running about *N* iterations each, the program has O(N^k) complexity.
- If your program is recursive with *b* recursive calls per level and has *l* levels, the program O(b^l) complexity.
- Bear in mind that there are *N!* permutations and 2^n subsets or combinations of *N* elements when dealing with those kinds of algorithms.
- The best times for sorting *N* elements are O(*N* log *N*).
- **DO THE MATH!** Plug in the numbers.

Examples

A single loop with *N* iterations is O(*N*):

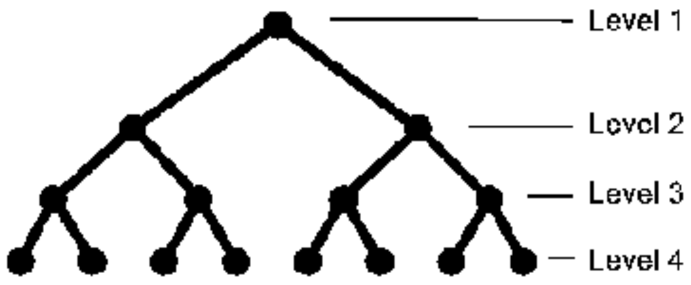
```
sum = 0
for i = 1 to n
  sum = sum + i
```

A double nested loop is often O(N^2):

```
# fill array a with N elements
for i = 1 to n-1
  for j = i + 1 to n
    if (a[i] > a[j])
       swap (a[i], a[j])
```

Note that even though this loop executes *N x (N+1) / 2* iterations of the if statement, it is O(N^2) since doubling *N* quadruples the execution times.

Consider this well balanced binary tree with four levels:



An algorithm that traverses a general binary tree will have complexity O(2^N).

Solution Paradigms

Generating vs. Filtering

Programs that generate lots of possible answers and then choose the ones that are correct (imagine an 8-queen solver) are *filters*. Those that hone in exactly on the correct answer without any false starts are *generators*. Generally, filters are easier (faster) to code and run slower. Do the math to see if a filter is good enough or if you need to try and create a generator.

Precomputation

Sometimes it is helpful to generate tables or other data structures that enable the fastest possible lookup of a result. This is called *precomputation* (in which one trades space for time). One might either compile precomputed data into a program, calculate it when the program starts, or just remember results as you compute them. A program that must translate letters from upper to lower case when they are in upper case can do a very fast table lookup that requires no conditionals, for example. Contest problems often use prime numbers - many times it is practical to generate a long list of primes for use elsewhere in a program.

Decomposition (The Hardest Thing At Programming Contests)

While there are fewer than 20 basic algorithms used in contest problems, the challenge of combination problems that require a combination of two algorithms for solution is daunting. Try to separate the cues from different parts of the problem so that you can combine one algorithm with a loop or with another algorithm to solve different parts of the problem independently. Note that sometimes you can use the same algorithm twice on different (independent!) parts of your data to significantly improve your running time.

Symmetries

Many problems have symmetries (e.g., distance between a pair of points is often the same either way you traverse the points). Symmetries can be 2-way, 4-way, 8-way, and more. Try to exploit symmetries to reduce execution time.

For instance, with 4-way symmetry, you solve only one fourth of the problem and then write down the four solutions that share symmetry with the single answer (look out for self-symmetric solutions which should only be output once or twice, of course).

Forward vs. Backward

Surprisingly, many contest problems work far better when solved backwards than when using a frontal attack. Be on the lookout for processing data in reverse order or building an attack that looks at the data in some order or fashion other than the obvious.

Simplification

Some problems can be rephrased into a somewhat different problem such that if you solve the new problem, you either already have or can easily find the solution to the original one; of course, you should solve the easier of the two only. Alternatively, like induction, for some problems one can make a small change to the solution of a slightly smaller problem to find the full answer.

Prime Cryptarithm

The following cryptarithm is a multiplication problem that can be solved by substituting digits from a specified set of N digits into the positions marked with *. If the set of prime digits {2,3,5,7} is selected, the cryptarithm is called a PRIME CRYPTARITHM.

```
      * * *
   x    * *
    -------
      * * *         <-- partial product 1
    * * *           <-- partial product 2
    -------
    * * * *
```

Digits can appear only in places marked by `*'. Of course, leading zeroes are not allowed.

The partial products must be three digits long, even though the general case (see below) might have four digit partial products.

********** Note About Cryptarithm's Multiplication ************
In USA, children are taught to perform multidigit multiplication as described here. Consider multiplying a three digit number whose digits are 'a', 'b', and 'c' by a two digit number whose digits are 'd' and 'e':

```
[Note that this diagram shows far more digits in its results than
the required diagram above which has three digit partial products!]

          a b c     <-- number 'abc'
        x   d e     <-- number 'de'; the 'x' means 'multiply'
     -----------
p1      * * * *     <-- product of e * abc; first star might be 0 (absent)
p2    * * * *       <-- product of d * abc; first star might be 0 (absent)
     -----------
      * * * * *     <-- sum of p1 and p2 (e*abc + 10*d*abc) == de*abc
```

Note that the 'partial products' are as taught in USA schools. The first partial product is the product of the final digit of the second number and the top number. The second partial product is the product of the first digit of the second number and the top number.

Write a program that will find all solutions to the cryptarithm above for any subset of supplied non-zero single-digits.

PROGRAM NAME: crypt1

INPUT FORMAT

Line 1:	N, the number of digits that will be used
Line 2:	N space separated non-zero digits with which to solve the cryptarithm

SAMPLE INPUT (file crypt1.in)

```
5
2 3 4 6 8
```

OUTPUT FORMAT

A single line with the total number of solutions. Here is the single solution for the sample input:

```
      2 2 2
    x   2 2
     ------
      4 4 4
    4 4 4
  ---------
    4 8 8 4
```

SAMPLE OUTPUT (file crypt1.out)

```
1
```

Submission file Name: 파일 선택 선택한 파일 없음 Send it in!

Combination Lock

Farmer John's cows keep escaping from his farm and causing mischief. To try and prevent them from leaving, he purchases a fancy combination lock to keep his cows from opening the pasture gate.

Knowing that his cows are quite clever, Farmer John wants to make sure they cannot easily open the lock by simply trying many different combinations. The lock has three dials, each numbered 1..N (1 <= N <= 100), where 1 and N are adjacent since the dials are circular. There are two combinations that open the lock, one set by Farmer John, and also a "master" combination set by the lock maker.

The lock has a small tolerance for error, however, so it will open even if the numbers on the dials are each within at most 2 positions of a valid combination.

For example, if Farmer John's combination is (1,2,3) and the master combination is (4,5,6), the lock will open if its dials are set to (1,3,5) (since this is close enough to Farmer John's combination) or to (2,4,8) (since this is close enough to the master combination). Note that (1,5,6) would not open the lock, since it is not close enough to any one single combination.

Given Farmer John's combination and the master combination, please determine the number of distinct settings for the dials that will open the lock. Order matters, so the setting (1,2,3) is distinct from (3,2,1).

PROGRAM NAME: combo

INPUT FORMAT:

Line 1:	The integer N.
Line 2:	Three space-separated integers, specifying Farmer John's combination.
Line 3:	Three space-separated integers, specifying the master combination (possibly the same as Farmer John's combination).

SAMPLE INPUT (file combo.in):

```
50
1 2 3
5 6 7
```

INPUT DETAILS:

Each dial is numbered 1..50. Farmer John's combination is (1,2,3), and the master combination is (5,6,7).

OUTPUT FORMAT:

Line 1: The number of distinct dial settings that will open the lock.

SAMPLE OUTPUT (file combo.out):

```
249
```

SAMPLE OUTPUT EXPLANATION

Here's a list:

```
1,1,1  2,2,4  3,4,2  4,4,5  5,4,8  6,5,6  7,5,9  3,50,2  50,1,4
1,1,2  2,2,5  3,4,3  4,4,6  5,4,9  6,5,7  7,6,5  3,50,3  50,1,5
1,1,3  2,3,1  3,4,4  4,4,7  5,5,5  6,5,8  7,6,6  3,50,4  50,2,1
1,1,4  2,3,2  3,4,5  4,4,8  5,5,6  6,5,9  7,6,7  3,50,5  50,2,2
1,1,5  2,3,3  3,4,6  4,4,9  5,5,7  6,6,5  7,6,8  49,1,1  50,2,3
1,2,1  2,3,4  3,4,7  4,5,5  5,5,8  6,6,6  7,6,9  49,1,2  50,2,4
1,2,2  2,3,5  3,4,8  4,5,6  5,5,9  6,6,7  7,7,5  49,1,3  50,2,5
1,2,3  2,4,1  3,4,9  4,5,7  5,6,5  6,6,8  7,7,6  49,1,4  50,3,1
1,2,4  2,4,2  3,5,5  4,5,8  5,6,6  6,6,9  7,7,7  49,1,5  50,3,2
1,2,5  2,4,3  3,5,6  4,5,9  5,6,7  6,7,5  7,7,8  49,2,1  50,3,3
1,3,1  2,4,4  3,5,7  4,6,5  5,6,8  6,7,6  7,7,9  49,2,2  50,3,4
1,3,2  2,4,5  3,5,8  4,6,6  5,6,9  6,7,7  7,8,5  49,2,3  50,3,5
1,3,3  3,1,1  3,5,9  4,6,7  5,7,5  6,7,8  7,8,6  49,2,4  50,4,1
1,3,4  3,1,2  3,6,5  4,6,8  5,7,6  6,7,9  7,8,7  49,2,5  50,4,2
1,3,5  3,1,3  3,6,6  4,6,9  5,7,7  6,8,5  7,8,8  49,3,1  50,4,3
1,4,1  3,1,4  3,6,7  4,7,5  5,7,8  6,8,6  7,8,9  49,3,2  50,4,4
1,4,2  3,1,5  3,6,8  4,7,6  5,7,9  6,8,7  1,50,1 49,3,3  50,4,5
1,4,3  3,2,1  3,6,9  4,7,7  5,8,5  6,8,8  1,50,2 49,3,4  49,50,1
1,4,4  3,2,2  3,7,5  4,7,8  5,8,6  6,8,9  1,50,3 49,3,5  49,50,2
1,4,5  3,2,3  3,7,6  4,7,9  5,8,7  7,4,5  1,50,4 49,4,1  49,50,3
2,1,1  3,2,4  3,7,7  4,8,5  5,8,8  7,4,6  1,50,5 49,4,2  49,50,4
2,1,2  3,2,5  3,7,8  4,8,6  5,8,9  7,4,7  2,50,1 49,4,3  49,50,5
2,1,3  3,3,1  3,7,9  4,8,7  6,4,5  7,4,8  2,50,2 49,4,4  50,50,1
2,1,4  3,3,2  3,8,5  4,8,8  6,4,6  7,4,9  2,50,3 49,4,5  50,50,2
2,1,5  3,3,3  3,8,6  4,8,9  6,4,7  7,5,5  2,50,4 50,1,1  50,50,3
2,2,1  3,3,4  3,8,7  5,4,5  6,4,8  7,5,6  2,50,5 50,1,2  50,50,4
2,2,2  3,3,5  3,8,8  5,4,6  6,4,9  7,5,7  3,50,1 50,1,3  50,50,5
2,2,3  3,4,1  3,8,9  5,4,7  6,5,5  7,5,8
```

Submission file Name: 파일 선택 선택한 파일 없음 Send it in!

USA COMPUTING OLYMPIAD

Wormholes

Farmer John's hobby of conducting high-energy physics experiments on weekends has backfired, causing N wormholes (2 <= N <= 12, N even) to materialize on his farm, each located at a distinct point on the 2D map of his farm (the x,y coordinates are both integers).

According to his calculations, Farmer John knows that his wormholes will form N/2 connected pairs. For example, if wormholes A and B are connected as a pair, then any object entering wormhole A will exit wormhole B moving in the same direction, and any object entering wormhole B will similarly exit from wormhole A moving in the same direction. This can have rather unpleasant consequences.

For example, suppose there are two paired wormholes A at (1,1) and B at (3,1), and that Bessie the cow starts from position (2,1) moving in the +x direction. Bessie will enter wormhole B [at (3,1)], exit from A [at (1,1)], then enter B again, and so on, getting trapped in an infinite cycle!

```
   | . . . .
   | A > B .      Bessie will travel to B then
   + . . . .      A then across to B again
```

Farmer John knows the exact location of each wormhole on his farm. He knows that Bessie the cow always walks in the +x direction, although he does not remember where Bessie is currently located.

Please help Farmer John count the number of distinct pairings of the wormholes such that Bessie could possibly get trapped in an infinite cycle if she starts from an unlucky position. FJ doesn't know which wormhole pairs with any other wormhole, so find all the possibilities.

PROGRAM NAME: wormhole

INPUT FORMAT:

Line 1:	The number of wormholes, N.
Lines 2..1+N:	Each line contains two space-separated integers describing the (x,y) coordinates of a single wormhole. Each coordinate is in the range 0..1,000,000,000.

SAMPLE INPUT (file wormhole.in):

```
4
0 0
1 0
1 1
0 1
```

INPUT DETAILS:

There are 4 wormholes, forming the corners of a square.

OUTPUT FORMAT:

Line 1:	The number of distinct pairings of wormholes such that Bessie could conceivably get stuck in a cycle walking from some starting point in the +x direction.

SAMPLE OUTPUT (file wormhole.out):

```
2
```

OUTPUT DETAILS:

If we number the wormholes 1..4 as we read them from the input, then if wormhole 1 pairs with wormhole 2 and wormhole 3 pairs with wormhole 4, Bessie can get stuck if she starts anywhere between (0,0) and (1,0) or between (0,1) and (1,1).

```
   | . . . .
   4 3 . . .      Bessie will travel to B then
   1-2-.-.-.      A then across to B again
```

Similarly, with the same starting points, Bessie can get stuck in a cycle if the pairings are 1-3 and 2-4 (if Bessie enters WH#3 and comes out at WH#1, she then walks to WH#2 which transports here to WH#4 which directs her towards WH#3 again for a cycle).

Only the pairings 1-4 and 2-3 allow Bessie to walk in the +x direction from any point in the 2D plane with no danger of cycling.

Submission file Name: 파일 선택 선택한 파일 없음 Send it in!

Ski Course Design

Farmer John has N hills on his farm (1 <= N <= 1,000), each with an integer elevation in the range 0 .. 100. In the winter, since there is abundant snow on these hills, FJ routinely operates a ski training camp.

Unfortunately, FJ has just found out about a new tax that will be assessed next year on farms used as ski training camps. Upon careful reading of the law, however, he discovers that the official definition of a ski camp requires the difference between the highest and lowest hill on his property to be strictly larger than 17. Therefore, if he shortens his tallest hills and adds mass to increase the height of his shorter hills, FJ can avoid paying the tax as long as the new difference between the highest and lowest hill is at most 17.

If it costs x^2 units of money to change the height of a hill by x units, what is the minimum amount of money FJ will need to pay? FJ can change the height of a hill only once, so the total cost for each hill is the square of the difference between its original and final height. FJ is only willing to change the height of each hill by an integer amount.

PROGRAM NAME: skidesign

INPUT FORMAT:

Line 1:	The integer N.
Lines 2..1+N:	Each line contains the elevation of a single hill.

SAMPLE INPUT (file skidesign.in):

```
5
20
4
1
24
21
```

INPUT DETAILS:

FJ's farm has 5 hills, with elevations 1, 4, 20, 21, and 24.

OUTPUT FORMAT:

The minimum amount FJ needs to pay to modify the elevations of his hills so the difference between largest and smallest is at most 17 units.

Line 1:

SAMPLE OUTPUT (file skidesign.out):

```
18
```

OUTPUT DETAILS:

FJ keeps the hills of heights 4, 20, and 21 as they are. He adds mass to the hill of height 1, bringing it to height 4 (cost = 3^2 = 9). He shortens the hill of height 24 to height 21, also at a cost of 3^2 = 9.
