GB21802 - Programming Challenges

Week 1 - What is Programming Challenges?

Claus Aranha
caranha@cs.tsukuba.ac.jp

Department of Computer Science

(last updated: April 9, 2021)

Version 2021.1

Week 1 – Part 1: Class Introduction

Outline

Part I: Class Summary

- In this class, you will practice your algorithm skills by writing many programs;
- Key Idea: Solve challenges with programs
  - Read the problem and choose the right algorithm;
  - Choose the implementation and data structure;
  - Run the program and check the correct result;
- In the 1st and 2nd year, we learn the theory of many algorithm. In this lecture, we learn the practice of these algorithms.

Algorithms: Theory x Implementation

It is very different to implement an algorithm in the classroom, and to implement an algorithm to solve a problem:

- Data Structure: You need to implement the function to read the input and store it in a data structure;
- Special Cases: Special cases in the input can cause bugs or make the problem more complex;
- Large Input: If the input is very large, the implementation needs to be efficient, or the program will run forever;
- Debugging: It can be hard to debug if the input is very hard, you need to learn how to make a test input;

Automated Judging

In this lecture we use **Automated Judges (AJ)** to check if your homework is correct.

- An AJ is a website that proposes programming puzzles, and receive solutions;
  - Ex: AtCoder, Aizu Online Judge, Topcoder, Codeforces, etc.
- The AJ receives your program; compile it; and grade it;
- The AJ test your program with a set of Hidden Inputs
- The AJ gives you the result: Correct or Incorrect
  - If the result is incorrect, it will not tell you why;
  - You have to create debug cases by yourself;
  - But, you can submit a new version as many times as you want;

What this class expects of you

Estimated classwork:

- You need basic programming knowledge in C++;
- Complete 2-4 program assignments per week;
  - Average of 4 hours of study per week;
  - A lot of time with debug and creating test cases;
  - (Atcoder difficulty: 300 to 500)
- Homework starts easy, but becomes harder later in the course;
- No final exam: only assignments!

Hint: Do your homework early!

What kind of problem is a "Programming Challenge?"

A "Programming Challenge" gives you a problem, and you have to write a program to find the solution. Think of it as a **programming puzzle**:

Simple Example

You want to schedule a pair dance club. Your friends give you their possible dates and times (Fri 14:00-15:00). What time do you choose to make the maximum number of pairs?

- Usually Programming challenges have a "story";
- There is a maximum execution time, so you have to use an efficient algorithm;
- Sometimes there are special cases;
- In general, a small program (< 200 lines) can solve the challenge;

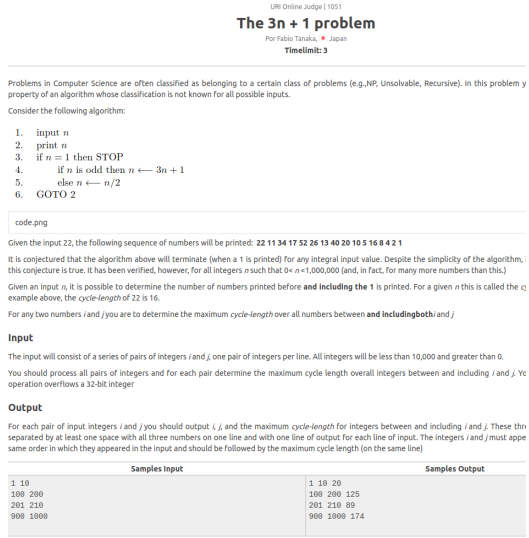
Why programming challenges?

- **For Competitions**: Around 1980, Programming Contests started as a way for Computer Sciences universities to compete. (IOI, ICPC, etc)
- **For Study**: After 2000, many people use online Automated Judges to study programming and prove their ability; (TopCoder, Codeforces, AtCoder, etc);
- **For Recruitment**: Recently, many companies use Programming Challenges to test the programming ability of candidates. candidates. (Google, Facebook, etc)
- **For Fun**: Some people just like to have an excuse to program!

Real Example (from homework): The 3n+1 problem

Contents of a Programming Challenge:

- Problem Description;
- Input Description;
- Output Description;
- Input/Output Example;



URI Online Judge | 1051

The 3n + 1 problem

Por Fabio Tanaka, Japan

Timelimit: 3

Problems in Computer Science are often classified as belonging to a certain class of problems (e.g.,NP, Unsolvable, Recursive). In this problem y property of an algorithm whose classification is not known for all possible inputs.

Consider the following algorithm:

```
1. input n
2. print n
3. if n = 1 then STOP
4.       if n is odd then n ⟵ 3n + 1
5.       else n ⟵ n/2
6. GOTO 2
```

code.png

Given the input 22, the following sequence of numbers will be printed: **22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1**

It is conjectured that the algorithm above will terminate (when a 1 is printed) for any integral input value. Despite the simplicity of the algorithm, i this conjecture is true. It has been verified, however, for all integers *n* such that 0< *n* <1,000,000 (and, in fact, for many more numbers than this.)

Given an input *n*, it is possible to determine the number of numbers printed before **and including the 1** is printed. For a given *n* this is called the cy example above, the *cycle-length* of 22 is 16.

For any two numbers *i* and *j* you are to determine the maximum *cycle-length* over all numbers between **and including both** *i* and *j*

Input

The input will consist of a series of pairs of integers *i* and *j*, one pair of integers per line. All integers will be less than 10,000 and greater than 0.

You should process all pairs of integers and for each pair determine the maximum cycle length overall integers between and including *i* and *j*. Yo operation overflows a 32-bit integer

Output

For each pair of input integers *i* and *j* you should output *i*, *j*, and the maximum *cycle-length* for integers between and including *i* and *j*. These thr separated by at least one space with all three numbers on one line and with one line of output for each line of input. The integers *i* and *j* must appe same order in which they appeared in the input and should be followed by the maximum cycle length (on the same line)

Samples Input	Samples Output
1 10 100 200 201 210 900 1000	1 10 20 100 200 125 201 210 89 900 1000 174

Real Example (from homework): The 3n+1 problem

How to solve?

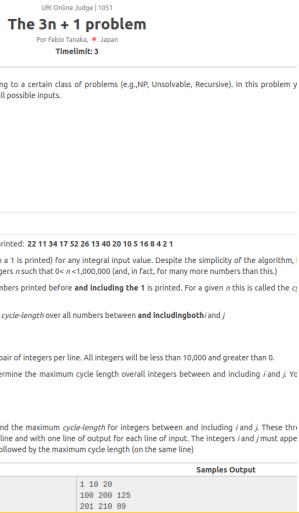
The problem wants the longest sequence size generated by the following algorithm:

1. if $n = 1$ then STOP
2. if n is odd, then $n = 3n + 1$
3. else $n = n/2$
4. GOTO 1

For example, if $i = 1$ and $j = 4$:

- n = 1: 1 END; **Length 1**
- n = 2: 2 1 END; **Length 2**
- n = 3: 3 10 5 16 8 4 2 1 END; **Length 8**
- n = 4: 4 2 1 END; **Length 3**

So the **maximum length** is 8 (for n = 3)



URI Online Judge | 1051

The 3n + 1 problem

Por Fabio Tanaka, Japan

Timelimit: 3

Problems in Computer Science are often classified as belonging to a certain class of problems (e.g.,NP, Unsolvable, Recursive). In this problem y property of an algorithm whose classification is not known for all possible inputs.

Consider the following algorithm:

```
1. input n
2. print n
3. if n = 1 then STOP
4.       if n is odd then n ⟵ 3n + 1
5.       else n ⟵ n/2
6. GOTO 2
```

code.png

Given the input 22, the following sequence of numbers will be printed: **22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1**

It is conjectured that the algorithm above will terminate (when a 1 is printed) for any integral input value. Despite the simplicity of the algorithm, this conjecture is true. It has been verified, however, for all integers *n* such that 0< *n* <1,000,000 (and, in fact, for many more numbers than this.)

Given an input *n*, it is possible to determine the number of numbers printed before **and including the 1** is printed. For a given *n* this is called the c example above, the *cycle-length* of 22 is 16.

For any two numbers *i* and *j* you are to determine the maximum *cycle-length* over all numbers between **and includingboth** *i* and *j*

Input

The input will consist of a series of pairs of integers *i* and *j*, one pair of integers per line. All integers will be less than 10,000 and greater than 0.

You should process all pairs of integers and for each pair determine the maximum cycle length overall integers between and including *i* and *j*. Yo operation overflows a 32-bit integer

Output

For each pair of input integers *i* and *j* you should output *i*, *j*, and the maximum *cycle-length* for integers between and including *i* and *j*. These thr separated by at least one space with all three numbers on one line and with one line of output for each line of input. The integers *i* and *j* must appe same order in which they appeared in the input and should be followed by the maximum cycle length (on the same line)

Samples Input	Samples Output
1 10 100 200 201 210	1 10 20 100 200 125 201 210 89

Real Example (from homework): The 3n+1 problem

A simple program

```
int main() {
  int min, max;
  int maxcycle = 0;
  cin >> min >> max;
  for (int i = min; i <= max; i++) {
    int cycle = 1;
    int n = i;
    while (n != 1) {
      if (n % 2 == 0) { n = n / 2; }
      else { n = n*3 + 1; }
      cycle++;
    }
    if (cycle > maxcycle) maxcycle = cycle;
  }
  cout << min << " " << max << " " << maxcycle << "\n";
  return 0;
}
```

Real Example (from homework): The 3n+1 problem

Simple programs, simple problems

The solution in the last slide has some problems. One problem is that it can be very slow! Consider the following case:

i = 1, j = 10:

- n = 1: 1 END
- n = 2: 2 1 END ...
- n = 7: 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 END

  ...
- n = 9: 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 END
- n = 10: 10 5 16 8 4 2 1 END

A lot of work is repeated without necessity. How to make the program faster?

Real Example (from homework): The 3n+1 problem

Memoization

A technique called **Memoization** can solve the "repeated work" problem.

Memoization:

- Every time you finish a calculation, store the result in the memory;
- Before you begin a calculation, check if it is not in the memory;

This technique can reduce the amount of repeated work.

In this course we will review and study many techniques like this one. You will have to implement these techniques in the homework to make it efficient.

Topics in this class

1. Introduction Problems
2. Data Structures
3. Search Problems
4. Dynamic Programming
5. Graphs Problems (Graph Structure)
6. Graph Problems (Graph Search and Flow)
7. String Manipulation
8. Math Problems
9. Geometry Problems
10. Final Remix

Class Format

Weekly Schedule

- Lecture Notes and Videos: Explanation about algorithms;
- TEAMS meeting with instructor to ask questions;
- Solve the homework at URI Online Judge;

Evaluation

- No final examination;
- Grade based on number of homework completed every week;

About the Lecturer

- Name: Claus Aranha;
- Country: Brazil;
- Research Topics:
  - Evolutionary Computation;
  - Artificial Life;
- Hobbies:
  - Game Programming;
  - Geocaching;
- webpage:
  `http://conclave.cs.tsukuba.ac.jp`

Extra: Join the Tsukuba ICPC Team!

What is ICPC?

If you like these contests, and want an extra challenge, please consider joining the Tsukuba ICPC team!

ICPC (International Collegiate Programming Contest) is the largest and most traditional programming competition between universities.

More than 50.000 students from all over the world participate in this competition every year.

Extra: Join the Tsukuba ICPC Team!

Program and see the world!

- Requirements: Team of 3 students, any course;
- Schedule:
  - National Preliminary Competition in July
  - Japanese Regional Competition in October
  - Asian Semi-final in December
  - World Final April next year

(Dates may change this year because of nCov-19)

Week 1 – Part 2: How this lecture is organized

Outline

What you will do every week

(manaba) Get the week PDF and study the lecture;

(manaba) Watch the lecture video;

(URI) Check the Programming Homework Exercises;

(TEAMS) Ask questions to the professor;

(URI) Submit your programs at the URI page;

(manaba) Complete the Homework Survey;

Class Dates and Deadlines

Class Dates

- 4/13, 4/20, 4/27, 5/11, 5/18, 5/25, 6/1, 6/8, 6/15, 6/22;
- No final exam;
- On-demand lecture + Office hours on Tue-34;

Deadlines

- The deadline for homework: **Monday every week**
- The deadline for late homework: 7/5 (one extra week)
- Final Grades will be published around 7/9

Dates subject to changes.

Where to find the material?

manaba and streams

- Official place for lecture materials and links;
- Use Forum for questions;
- Please read announcements; Please answer surveys;
- Self-registration Code for non-credit students: 4754254

github

- Lecture materials is also available on github:
- URL: `https://caranha.github.io/Programming-Challenges/`
- Not-official. Includes material from last year.
- Access if manaba is not available.

Websites for the programming homework

URI Online Judge – check / submit howework here

- `https://www.urionlinejudge.com.br`
- Discipline **ID: 007272** (name: Programming Challenges, Spring 2021)
- Key: **B8xVp90**

- **Important!**
- After you create your account, submit the "ID survey" in manaba;
- After you complete each homework, submit the "Homework survey" in manaba;

For more information about how to use URI, please see the "URI Howto" video, or read the outline on manaba

About Course Language

Natural Language

- Weekly Materials and Homework: English
- Weekly Video and manaba: Japanese
- E-mail, feedback: Any language you want;
- If you want to help me translate the homework, contact me!

Programming Language

- Officially, we only support C and C++;
- The Judge accepts: C, C++, Java, Python;
- If you want to use another language, contact me; No promises.

Reference Books

Textbook:

- **textbook:** Steven Halim, Felix Halim,"Competitive Programming", 3rd edition. `https://cpbook.net/`

Other references:

- Steven S. Skiena, Miguel A. Revilla,"Programming Challenges", Springer, 2003
- 秋葉拓哉、 岩田陽一、 北川宜稔,『プログラミングコンテストチャレンジブック』
- 渡部有隆、『オンラインチャレンジではじめるC/C++プログラミング入門、Online Programming Challenge!』 (ISBN978-4-8399-5110-8)
- 渡部有隆、『プログラミングコンテスト攻略のためのアルゴリズムとデータ構造』（ISBN978-4-8399-5295-2）

Grading Rules

Base Grade

Your **base grade** is based on the number of accepted homework programs you submit:

- **A grade:** 4+ accepted minimum every week;
- **B grade:** 3+ accepted minimum every week;
- **C grade:** 2+ accepted minimum every week;
- **D grade:** less than 2 accepted every week.

Important!!

- Only "Accepted" programs count. "Wrong Answer", "Time limit" does not count;
- The number above is **minimum every week**. NOT AVERAGE.

Grading Rules

Best Bonus

For each grade group, 10% of the students with the **most total problems** receive a bonus grade. The bonus grade raise 1 step: C to B, B to A, A to A+, etc.

Example:

15 students are in base grade A (4 problems per week):

- 8 students with 40 problems
- 1 student with 41 problems
- 2 students with 50 problems
- 1 student with 65 problems
- 3 students with 72 problems

The 3 students with 72 problems increase their grade from A to A+.

Grading Rules

Late Penalty

If you do not meet the deadline, you can submit your homework after the deadline. If you submit too many programs after the deadline, you will receive a grade penalty.

If the number of late programs $\geq$ 25% of total programs, your grade will lower 1 step.

You will not fail the course for late programs.

Example:

- Student (1) submitted 40 problems, minimum 4 problems per exercise. 5 problems are late. $5 \leq 40 * 0.25$, no penalty. Grade A.
- Student (2) submitted 44 problems, minimum 4 problems per exercise. 16 problems are late. $16 \geq 44 * 0.25$, **penalty**. Grade B.
- Student (3) submitted 24 problems, minimum 2 problems per exercise. 10 problems are late. $10 \geq 24 * 0.25$, **penalty**. Grade C (will not fail).

Grading

Plagiarism

The assignments are individual. You must write your programs by yourself.

You can do this

- Ask for ideas to your friends;
- Ask for ideas in the MANABA forum;
- Ask for help with a bug;

You can NOT do this

- Copy a solution from the internet;
- Copy a solution from your friends;
- Give your code to a friend;

Students who do plagiarism will fail the course, and suffer penalties from the university.

Week 1 – Part 3: AdHoc Problems

What are Ad-hoc problems?

Ad-hoc means "single purpose". In programming challenges, it usually means "simple problem" or "Follow the instructions".

This week, we will some some simple problems, and learn skills that are useful for more complex problems too:

- How to find important information in the problem text;
- How to understand the input and output;
- Generating input data for debugging;
- etc;

Today's lecture focus on hints that can be used the entire course.

Revisiting the 3n+1 problem

Let's revisit the 3n+1 problem, that we introduced in the beginning of this class.

This is a great problem because it is very simple, but you still need to be careful when implementing the solution.

Revisiting the 3n+1 problem

Problem outline

Calculate the **Maximum Cycle Length** between any two numbers *i* and *j*.

```
Cycle of (n)
1. print n
2. if n == 1 then STOP
3. if n is odd then n = 3n + 1
4. else n = n/2
5. GOTO 2

Example: Cycle of (22)
22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

Size: 16
```

A simple solution for the 3n+1 problem

```
while true:
  try:
    line = input()
    max = 0
    tk = line.split()
    i, j = int(tk[0]), int(tk[1])
    for n in range(i, j+1):
      count = 1
      while n != 1:
        if n % 2 == 1: n = 3 * n + 1
        else: n = n / 2
        count += 1
      if count > max: max = count
    print (line, max)
  except EOFError: break
```

A Simple Solution has a Simple problem

- If you submit the program in the previous slide, you will receive the result Wrong Answer from the judge.
- But the problem gets the correct input for the sample input!
- Even if you try a few more i and j, it is still correct.
- What is wrong with the simple solution? (Answer in the next slide)

Trick Hidden Inputs

- The first solution solves all the sample input correctly, but there are some inputs that can cause problems.

- For example, think of this input: $i = 20, j = 10$
- The output will be nothing!

- Here is the code that causes this problem:

```
for x in range(i, j+1):    <-- Error is here!
  ...
  print (line, max)
```

- But in the input there are no examples with "$i > j$" is this allowed?

Trick Hidden Inputs

Reading the input carefully

The problem page has a description that include an **input** section:

Input Description

The input will consist of a series of pairs of integers i and j, one pair of integers per line. All integers will be less than 10,000 and greater than 0.

You should process all pairs of integers and for each pair determine the maximum cycle length over all integers between and including i and j.

You can assume that no operation overflows a 32-bit integer.

In the input section, there is no rule that forces $j > i$!

If it is not forbidden by the rules, then maybe it will happen!

Trick Hidden Inputs

General Rules

- First rule of Programming Challenges:
  If it is not written, assume the worst case
- Some common tricky inputs to be careful:
  - A number is negative; a number is zero; a number is maximum;
  - The input is out of order (or in order!);
  - The input is repeated;
  - A graph is unconnected; a graph is fully connected;
  - Lines are parallel; points are in the same place;
  - An area is 0; an angle is 0;
  - The input is very long;
  - The input is very short;
- If it is not against the rules, it may (will?) happen;
- This first rule is also important for the real world!

Problem 3n+1

A fixed solution for the input

```
while true:
  try:
    line = input()
    max = 0
    tk = line.split()
    i, j = int(tk[0]), int(tk[1])
    for n in range(min(i, j), max(i, j)+1): # FIXED!
      count = 0
      while n != 1:
        if n % 2 == 1: n = 3 * n + 1
        else: n = n / 2
        count += 1
      if count > max: max = count
    print (line, max)
  except EOFError: break
```

This solution has a problem too!

- This time we are sure that the solution is correct for all inputs;
- But you still do not get "accepted" in the online judge...
- Now the judge says "Time limited exceeded".
- What happened? (Think a little bit on this one)

Solution 2: Be careful of computing costs!

Input Description

All integers will be less than 10,000 and greater than 0.

You can assume that no operation overflows a 32-bit integer.

- What happens when the input is minimum and maximum?

```
1 10000 262
```

- A sequence with 262 steps does not seem very long.
- But remember we calculate all sequences between 1, 10000!
- Estimate the cost: $10000 \times 262 \approx 2,000,000$ steps!

- This is one query. The input can have repeated queries!

Avoiding repeated work

- Think about the cycle calculation (let's call it A(n));

```
A(22):  22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
A(11):  11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
A(17):  17 52 26 13 40 20 10 5 16 8 4 2 1
A(13):  13 40 20 10 5 16 8 4 2 1
A(10):  10 5 16 8 4 2 1
A(8) :  8 4 2 1
```

- Can we avoid all this recalculation?

Good Technique: Memoization

Store the result of a function that we know will be calculated again in the future. This is useful for many problems!

Avoiding Repeated Work

Memoization Idea

To implement Memoization, we use a table or dictionary to store partial results.

```
table = {}
table[1] = 1
def A(n):
  if n in table.keys(): return table[n]
  else:
    if n % 2 == 1: table[n] = 1 + A(3*n + 1)
    else: table[n] = 1 + A(n/2)
  return table[n]
```

In a future class, we will focus on Dynamic Programming, which generalizes this technique.

A Programming Challenge Workflow

When solving a programming challenge, you want to follow these common steps:

- Task 1: Read the problem description;
- Task 2: Read the input/output;
- Task 3: Think about the algorithm;
- Task 4: Write the Code;
- Task 5: Test the program on example data;
- Task 6: Test the program on hidden data;

Task 1: Understanding the Problem Description

Reading and understanding the problem is very important! Try to understand completely the problem before starting to program.

General Tips

- Find the rules of the problem, separate from the story;
- Sometimes it is easy to read the input/output part first;
- Problems with a lot of story are usually not very hard.;

Reading English is Hard!

- Don't worry!
- Focus on the keywords;
- Understanding a little English is important for CS!
- Get help from professor Google Translate;

Example: Problem 11559 – Event Planning

Story:

As you didn't show up to the yearly general meeting of the Nordic Club of Pin Collectors, you were unanimously elected to organize this years excursion to Pin City. You are free to choose from a number of weekends this autumn, and have to find a suitable hotel to stay at, preferably as cheap as possible.

Rules

You have some constraints: The total cost of the trip must be within budget, of course. All participants must stay at the same hotel, to avoid last years catastrophe, where some members got lost in the city, never being seen again.

How do you tell the difference between Story and Rules? Look for the keywords!

Keywords: constraints, minimum, maximum, cost, rules, number, etc...

Hints for Reading Problems

- First, look at the sample input and output;
- Write the key ideas on paper
- Use the Paper: mark keywords;
- Use the Paper: cut flavor;
- Read the problem again!;
- Do not begin programming until you understand the problem!

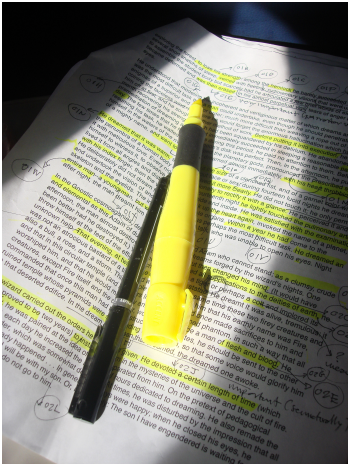


Image by Guido “random” Alvarez, released as CC-BY-2.0

Task 2: Reading the Input/Output

The Input Description is very important (as we saw in 3n+1)

- What is the stop condition?
  - The number of inputs is given in the beginning;
  - Special value to indicate the end of the input;
  - Input ends at the end of the file (EOF);
- What is the format of the input/output?
  - Very important for output (floating point precision)
- What is the size of the input?
  - What is the maximum number of inputs?
  - What are the maximum and minimum values?
  - Are there special conditions?

Task 2: Estimate time limit from input size

The input size shows how big the problem gets.

- Small Problem: You can use brute force algorithms;
- Big Problem: You need to use more complex algorithms; Maybe prune the input;

Estimate the "Time Limit Exceeded" limit

- expect around 10,000,000 operations per second on the online judge;
- most problems have 1-3 seconds of time limit;
- python can be a bit slower!

Task 2: Input size examples

n < 24

Exponential algorithms will work ($O(2^n)$). Or just calculate all solutions.

n = 500

Cubic algorithms don't work anymore ($O(n^3) = 125.000.000$)
Maybe $O(n^2 \log n)$ will still work.

n = 10.000

A square algorithm ($O(n^2)$) might still work. But beware any big constants!

n = 1.000.000

$O(n \log n) = 13.000.000$
We might need a linear algorithm!

Task 2: Input Format

Three common patterns for input format:

- Read N, then read N queries;
- Read until a special condition;
- Read until EOF;

Task 2: Input Format

Read *N*, and then read *N* queries;

Remember *N* when calculating the size of the problem!

Example: Cost Cutting

```
#include <iostream>
using namespace std;

int main() {
    int n; cin >> n;

    for (; n > 0; n--) {
       // Do something
    }
}
```

Task 2: Input Format

Read Until a Special Condition.
Be careful: You can have many queries before the condition!

Example: Request for Proposal:
The input ends with a line containing two zeroes.

```
int main()
{
   cin >> n >> p;
   while (n!=0 || p!=0)
   {
       // do something!
       cin >> n >> p;
   }
}
```

Task 2: Input Format

Read until EOF.

Functions in C and Java return FALSE when they read EOF. Python requires an exception. Very common in UVA.

Example: 3N+1 Problem, Jolly Jumpers

```
int main()
{
    int a, b;
    while (cin >> a >> b;)
    {
        // Do something!
    }
}
```

Task 2: Output Format

The UVA judge decides the result based on a simple diff.

Be very careful that the output is exactly right!

The Judge is like an angry client. It wants the output EXACTLY how it stated.

Task 2: Output Format – Checklist

1. DID YOU REMOVE DEBUG OUTPUT?
2. DID YOU REMOVE DEBUG OUTPUT?
3. Easy mistakes: UPPERCASE x lowercase, spleling mitsakes;
4. Boring mistakes: plural: 1 hour or 2 hours;
5. What is the precision of float? (3.051 or 3.05)
6. Round up or Round down? ($3.62 \rightarrow 3$ or 4)
7. Multiple solutions: Which one do you output (usually ortographical sort)

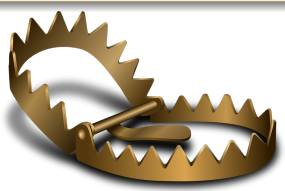
Task 2: Tricks in the input/output

Example: 3n+1 Problem

- *i* and *j* can come in any order.

Common Traps

- Negative numbers, zeros;
- Duplicated input, empty input;
- No solutions, multiple solutions;
- Other special cases;



Task 3: Choosing the algorithm

The important part of choosing the algorithm is counting the time

- An algorithm with k-nested loops of and n commands has $O(nk)$ complexity;
- A recursive algorithm with b recursive calls per level, and L levels, it should have $O(bL)$ complexity;
- An algorithm with p nested loops of size n is $O(n^p)$
- An algorithm processing a $n * n$ matrix in O(k) per cell runs in $O(kn^2)$ time.

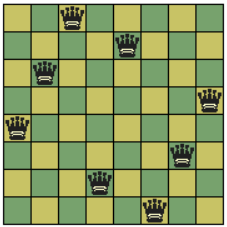
One way to reduce the cost of a program is to use pruning

Task 3: Example of Pruning – 8 queen problem

Pruning is when you quickly remove bad cases to reduce the computational cost of an algorithm.

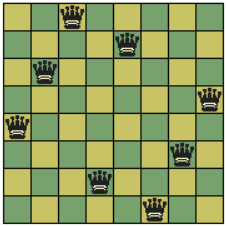
The "8-queens" problem is a good problem to understand pruning. We want to find all possible positions of 8 queens in a chessboard, where they don't attack each other.

The brute force algorithm is: Try all possible positions, and check if they are valid. But how we list all positions make a big difference.



Task 3: Example of Pruning – 8 queen problem

Approach 1 – all squares

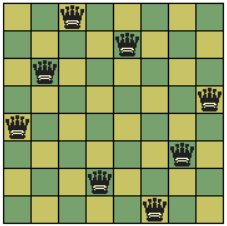


Approach 1: For each queen, test all possible squares;

```
Queen1: a1 or a2 or a3 or a4 ... or h5 or h6 or h7 or h8
Queen2: Same as Queen 1
Queen3: Same as Queen 2
...
Queen8: Same as Queen 7
```

Task 3: Example of Pruning – 8 queen problem

Approach 2 – columns

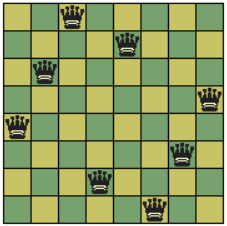


Approach 2: Choose one column for each queen, test all rows.

```
Queen1: a1 or a2 or a3 ...
Queen2: b1 or b2 or b3 ...
Queen3: c1 or c2 or c3 ...
...
Queen8: h1 or h2 or h3 ...
```

Task 3: Example of Pruning – 8 queen problem

Approach 3 – permutation



Choose one column for each queen, test a row not used.

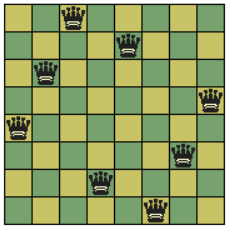
```
If Q1 is a1, Q2 in b2, Q3 must be c3-7...

A solution is the order of rows: Ex: 1-3-5-2-7-4-8-6

Total Solutions: 8 x 7 x 6 x 5 ... = 40320
```

Task 3: Example of Pruning – 8 queen problem

Pruning comparison



- Approach 1: All rows and columns: 10^{14} steps;
- Approach 2: All rows: 10^{7} steps;
- Approach 3: Permutation: 40320 steps;

Same algorithm (brute force), but pruning changes the efficiency!

Task 4: Coding

After you understand what the problem requires you to do, and after you decide which algorithm to use, then you can begin programming.

Avoid beging programming before you have a good idea of what you are going to do. Early programming might lead to bugs, and you may decide that you want to change your algorithm or data structure;

Hints when coding

Hint 1: Coding Library

As you solve many programs, save functions and patterns that you use many times:

- Code for different types of Input;
- Common data structures;
- Difficult algorithms;

Hint 2: Use paper

When you write your idea on paper (diagrams, code), it helps you clarify parts that are not well defined in your mind.

Sometimes this helps you find problems in your idea early.

Hints when Coding

Hint 3: "Programmer Efficiency"

We often think about **CPU efficiency** (program is fast), or **memory efficiency** (program uses little memory).

But another very important efficiency is **Programmer Efficiency**: If the program is too complex, the programmer will get tired or confused!

- Complex programs take longer to complete;
- Complex programs are harder to debug;
- You can't reuse code from complex programs easily;

How to improve programmer efficiency:

- Learn and use the standard library and macros;
- Create your own library of programming challenge patterns;
- Focus on the minimal features for this problem;

Sample data and Hidden Data

Remember that the sample data in the program is not all data! The Online Judge will use a set of **hidden data**.

Generate your own set of data to test your program before submitting.

Sample Data

- Useful to test the input/output functions;
- You can read the sample data to understand the problem;
- Does NOT include tricky cases;

Hidden Data

- Data of the online judge;
- Much bigger cases, uses maximum and minimum values;
- Data includes tricky cases and worst cases;

Generating testing data cases

"Be mean. Generate data that will show bugs in your program, not data that you expect to work."

- Repeat one case many times in the same file (check for initialization);
- Random data with maximum size/values (test for array limits and general performance);
- Border cases (maximum and minimum values);
- Worst cases (what data would make the algorithm run slowest? If you don't know the worst case, maybe you don't understand the algorithm yet!);

In particular for random and large test data sets, I recommend that you create a simple script that generates random data for you. Random data is great for finding "crash" bugs.

To Summarize

Mental framework to solve problems:

1. Read the problem carefully to avoid traps;
2. Think of the algorithm and data structure;
3. Keep the size of the problem in mind;
4. Keep your code simple;
5. Create special test cases;

Now go solve the other problems!

Thanks for Listening!

Send questions to the manaba forums!