NYU

CSCI-UA.0480-004
Algorithmic Problem Solving

Lecture 2
The Fundamentals

Bowen Yu
Spring 2018

- Piazza
  - Course announcement and discussion.
  - Signup: https://piazza.com/nyu/spring2018/csciua0480004
- Coda (HW and quizzes/exams)
  - signup instructions will be posted on Piazza
- Vjudge (recitation)
  - Create an account with username: {netID}_CS480S18
    - For example, if your netID is by123, your username is: by123_CS480S18
  - Apply to the group https://vjudge.net/group/aps-spring18

- Grades will be released on NYUClasses
  - Note that you can calculate the grades yourself as HW/Recitation/Exam scores can be directly found on coda/vjudge.
  - It is also used for receiving free study reports.

Homework Last Time

- Complete the logistics signup's as on previous slide.
- HW1 released
  - Two simple I/O questions
  - If you notice coda system issues, please report on Piazza with label "coda_issues".

Quizzes and Exams

- When
  - **Fridays, at recitation time!**
    - Except for the final that is scheduled otherwise by the department
  - Please see Piazza for dates

- Obligatory slide on Academic Integrity
- Collaboration Policy:
  - You can discuss homework and recitation questions with classmates.
  - You must write and submit your own solution.
    - You must never copy a single line of code from another student.
- Using outside resources:
  - You are free to consult online resources in general.
    - You must NOT search for solutions to homework/recitation questions.
  - You must never copy code online into your solution.
    - You may, after understanding the code, rewrite it yourself.
    - If you do so, you must properly cite the source by adding comments in the code
      - E.g. “I referred to http://some_url about the implementation of this technique...”
    - You must be able to explain every line of your work
  - You can use sample code released from the course without citation.
- We run strict anti-cheat tests on all your submissions. Do not try your “luck”!!

- This class is inspired by the ACM International Collegiate Programming Contest (ICPC)
- Problems exactly like the ones we will be studying
  - 5 hours to solve 9+ problems
- Team-based: 3 contestants per team with 1 computer
  - Forces you to think before you code
  - Must proofread to debug (machine time is limited)
- Scoring
  - Judging is also by black-box testing
  - Problems are weighted equally: 1 point for each problem regardless of its difficulty
  - Time penalty (20min for incorrect submission) for breaking ties
- Multi-tier: Regional → World Finals

“…32,043 contestants from 2,286 universities in 94 countries competed in regional competitions at over 300 sites worldwide…”

ACM ICPC fact sheet

NYU in ICPC

Photo@2014 World Finals, Ekaterinburg

2012-2013: 3rd in GNYR

2013-2014: 1st in GNYR
North American Champion in WF
(13th overall)

2014-2015: 8th in GNYR

2015-2016: 4th in GNYR

2016-2017: 3rd in GNYR

2017-2018: 7th in GNYR

Why teach a class motivated by a programming contest?

- The way that OJ works is very encouraging for people to learn programming and algorithms.
- Makes you a better programmer, thinker, and problem-solver.
- Prepares you for technical interviews
  - Many companies host their own programming contests for recruitment
- It’s fun!
  - There are many who compete as hobby.

Interested in Competition?

- **Join NYU Programming Team**
  - https://cs.nyu.edu/acm/progteam/
  - Signup for the Progteam mailing list to receive event notification.
    - https://cs.nyu.edu/mailman/listinfo/progteam
- **Progteam Events**
  - Friday Night Practice: code with pizza
    - Questions are good complement exercise for this course.
  - Algorithm Cyclone: small-group discussion of harder problems
  - ICPC Training
  - Competitive programming community

Practice Archive

- USACO online: http://www.usaco.org
  - Bronze, Silver, Gold, Platinum divisions
- Kattis: https://open.kattis.com/
  - High-quality problems from contests
- UVA Online Judge: http://uva.onlinejudge.org

Contests

- HackerRank: https://www.hackerrank.com/
- TopCoder: http://www.topcoder.com
- Codeforces: http://codeforces.com
- CodeChef: https://www.codechef.com/
- CSAcademy: https://csacademy.com/

Extra Credit Options

- Contest participation
  - Google Codejam (usually around April)
    - Passing qualification: +2 points
    - Passing first round: +3 points
    - Passing second round: + ?? points
  - NAIPC (North America Invitational Programming Contest)
    - A warmup contest for NA ICPC Finalists
    - With prizes
    - Team contest: +2 points to every team member, for every problem solved

How to Study this Course

- Follow closely, complete in time, and fully understand the lecture, homeworks and recitations material
  - Take stress here!
  - Lecture attendance is highly recommended
- For every problem
  - Do not look at or hear others' solutions before thinking about it yourself
  - Be brave to try your own idea (and be prepared to fail)
  - Understand every issue you are having
  - Always write your solutions independently
- Stay relaxed for quizzes and exams.

- **Asymptotic Complexity**
  - Nothing is more important than this!!!

- **Other Fundamentals**
  - Bits, bytes, and variable sizes
  - Integer in different bases
  - Lexicographical ordering
  - Modular arithmetic

- A formal way to assess how fast a program runs
  - Time complexity
    - related to Time Limit Exceeded (TLE)
  - Space complexity
    - related to Run Time Error (RE) or Memory Limit Exceeded (MLE)
- With complexity analysis in place, you should be able to tell that an idea is not efficient enough and will not work
  - So that you can think about other ideas
  - So that you can avoid writing solutions that are doomed to fail

- Operators
  - Arithmetic: +, -, *, /, %
  - Logical: &&, ||
  - Bitwise: &, |, ^
- Value access
  - arr[3]
  - obj.attr
- Assignment
  - x = 5
- Condition check
  - if else
  - loops: for, while
- I/O
  - reading input
  - printing

How Many Operations?

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6    for (int i = 0; i < 100; i++) {
7      for (int j = 0; j < 50; j++) {
8        printf("%d\n", i + j);
9      }
10   }
11   return 0;
12 }
```

How Many Operations? (cont'd)

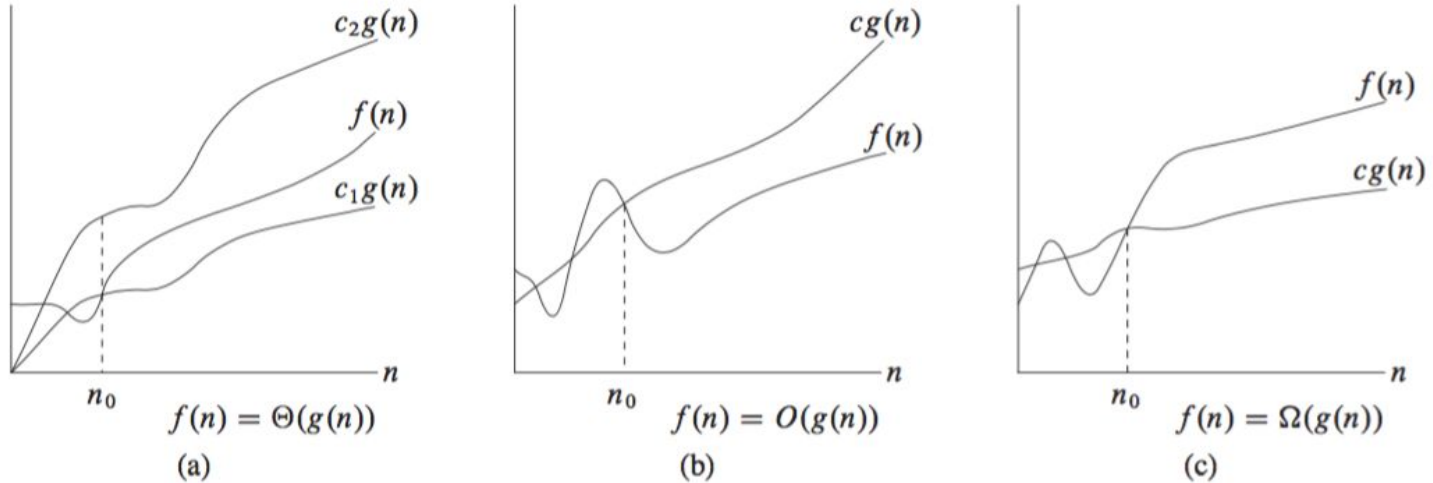
```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6    int ans = 0;
7    for (int i = 0; i < 100; i++) {
8      for (int j = 0; j < 50; j++) {
9        ans += i + j;
10       if (ans % 2) ans /= 2;
11     }
12   }
13   printf("%d\n", ans);
14   return 0;
15 }
```

Counting Operations is Tedious

- It is complicated and virtually impossible to count the exact number of operations a program will execute
- Some operations are faster than the others
  - addition is faster than multiplication
- We need a simplified way to quickly assess the efficiency of a program.

Just Care about the Magnitude

- Assume we are dealing with N input elements.
  - Do a linear scan / Create a copy of them
    - N operations
  - Check all pairs:
    - $N(N-1)/2 = N^2/2 - N/2$?
    - => simplified to N^2
    - We are ignoring constants and lower-order terms
- Examples
  - $2N^3 + 3N^2 + 5N \Rightarrow N^3$
  - $N^2 + N\log N \Rightarrow N^2$
  - $3*2^N + N^{100} \Rightarrow 2^N$
  - $N! + 100^N \Rightarrow N!$
  - $N*\sqrt{N} + N\log N \Rightarrow N*\sqrt{N} = N^{1.5}$



Refer to *Introduction to Algorithms*

Big-* Notations

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .^{1}$$

Small-* Notations

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\} .$$

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0\} .$$

- We write f(N) = O(g(N)) to denote membership.

- With APS, we use a convention that slightly differs from the formal definition.
  - You will hear something like “Your algorithm is $O(N^3)$ and too slow”
  - But a constant function f(N) = 1 is also $O(N^3)$.
  - Why? If you think about it, this is actually very confusing…

Clarification on Asymptotic Terms

When we say “Your algorithm is $O(N^3)$”, we actually mean:

- Theoretically, there exists some (category of) worst input on which your algorithm is $\Omega(N^3)$, and most likely also $\Theta(N^3)$.
- Practically speaking, there exists a breaker case on which your algorithm needs (asymptotically) N^3 operations.

However, as a conventional simplification, competitive programmers just use big-O to denote the worst-case scenario of an algorithm. In this course, we follow this convention.

- A typical modern workstation can execute about $10^{8\sim9}$ operations per second.
  - Again, some operations are slower than the others, but we usually treat them uniformly.
- To estimate the execution time of your program on worst-case scenario:
  - Evaluate the Big-O term using the maximum value of N (largest input size)
  - Then divide the result by 10^9
- Examples: Check all pairs $O(N^2)$
  - N=10: $N^2 = 100$, time = $100/10^9$ => finish instantly (0.00s)
  - N=1000: $N^2 = 10^6$, time = $10^6/10^9$ => finish almost instantly (<0.01s)
    - Note that there is always some overhead so it may not be 0.001s
  - $N=10^5$: $N^2 = 10^{10}$, time = $10^{10}/10^9$ => about 10 seconds (typically TLE)

Simple 3-level nested loops

```
1 for (int i = 0; i < n; i++) {
2   for (int j = i + 1; j < n; j++) {
3     for (int k = j + 1; k < n; k++) {
4       ans += i * j;
5       ans += j * k;
6       ans += k * i;
7     }
8   }
9 }
```

Enumerate sub-matrices in a matrix of size (n, m)

```
for (int x1 = 0; x1 < n; x1++) {
  for (int y1 = 0; y1 < m; y1++) {
    for (int x2 = x1; x2 < n; x2++) {
      for (int y2 = y1; y2 < m; y2++) {
        // ...
      }
    }
  }
}
```

Compute the sum of all sub-matrices in a matrix of size (n, m)

```
for (int x1 = 0; x1 < n; x1++) {
  for (int y1 = 0; y1 < m; y1++) {
    for (int x2 = x1; x2 < n; x2++) {
      for (int y2 = y1; y2 < m; y2++) {
        // begin summing sub-matrix
        for (int i = x1; i <= x2; i++) {
          for (int j = y1; j <= y2; j++) {
            ans += matrix[i][j];
          }
        }
        // end summing sub-matrix
      }
    }
  }
}
```

Check if a string Y is a substring of another string X

```
1  boolean substringMatch(String X, String Y) {
2    for (int i = 0; i < X.length; i++) {
3      boolean matched = true;
4      for (int j = 0; j < Y.length; j++) {
5        if (i + j >= X.length) matched = false;
6        if (X[i] != Y[j]) matched = false;
7      }
8      if (matched) return true;
9    }
10   return false;
11 }
```

It’s not that simple as just reading the number of nested loops!

unique(): shrinking consecutive same elements in an array into one
[1, 1, 3, 5, 5, 5, 2, 2, 7] => [1, 3, 5, 2, 7]

```
1 for (int i = 0; i < n; i++) {
2   int j = i;
3   while (j < n && oldArray[j] == oldArray[i]) j++;
4   newArray.push(oldArray[i]);
5   i = j - 1;
6 }
```

Two nested loops but it’s linear!

Check if a string Y is a substring of another string X (“optimized”)

```
1  boolean substringMatch(String X, String Y) {
2    for (int i = 0; i < X.length; i++) {
3      boolean matched = true;
4      for (int j = 0; j < Y.length; j++) {
5        if (i + j >= X.length) matched = false;
6        if (X[i] != Y[j]) matched = false;
7        if (!matched) break; // I break early!!!
8      }
9      if (matched) return true;
10   }
11   return false;
12 }
```

This won’t make you linear. Check from right to left also doesn’t work.

Breaker:

X = aaa...aaabaaa...aaa
Y = aaa...aaaa

Do Not Optimize Constant

- In practice, constant optimization is useful… e.g.
  - Can make a server respond to more users
  - 30FPS in a game is much better than 15FPS

- However in this course, constant optimization is discouraged.
  - Constant optimization is most of the time tedious and not very fun (faster I/O? :/)
  - It's not much about being algorithmic.
  - Well-designed algorithmic problems shall cleanly separate slow TLE solutions from AC ones.
    - Such scenario shall not happen for a problem with TL = 1 sec
      - You had a correct algorithm that somehow ran for 1.1s.
      - You scratched your head and guessed that standard library's ArrayList<Integer> is too slow… and you chose to go with native array implementation (int []).
      - Then it passed with 0.95s…
      - Later you found that author's solution also uses int[]...
    - We shall allow any reasonably well written solutions of a correct algorithm to pass.
      - But be aware that sometimes this may be hard to control.

Polynomial

- 1, logN, N, NlogN, $Nlog^2N$, N*sqrt(N), N^2, N^2logN, N^3, N^5
- Most algorithms in APS fall in this category.

Exponential

- 2^N, N!, C(N,M)
- Usually brute-forcing algorithms
  - Find a best subset out of N elements? Enumerate pick or not-pick for each element: $O(2^N)$

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!), O(n^6)$	e.g. Enumerating permutations (Section 3.2)
$\leq [15..18]$	$O(2^n \times n^2)$	e.g. DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g. DP with bitmask technique (Section 8.3.1)
≤ 100	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, ${}_nC_{k=4}$
≤ 400	$O(n^3)$	e.g. Floyd Warshall's (Section 4.5)
$\leq 2K$	$O(n^2 \log_2 n)$	e.g. 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$ 5000	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 1M$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree (Section 2.3)
$\leq 100M$	$O(n), O(\log_2 n), O(1)$	Most contest problem has $n \leq 1M$ (I/O bottleneck)

Table 1.4: Rule of thumb time complexities for the 'Worst AC Algorithm' for various single-test-case input sizes n, assuming that your CPU can compute $100M$ items in 3s.

Approaching 10^8 is dangerous!

Usually an expected solution is at most 10^7. Use your own judgment on the constant factor.

- 1 bit is binary (one or zero)
- 1 byte = 8 bits
- Computer hardware can only read/write bytes (not bits)

Primitive variable types

- boolean/char: 8-bit
- int: 32-bit
- long (Java) / long long (C++) : 64-bit
- float: 32-bit
- double: 64-bit

- String is NOT a primitive variable type
  - In terms of Object Oriented Programming (OOP), it is a class
- String is stored as an array of char's.
- Comparing two strings' equality is NOT constant time! (actually linear in terms of string length)

Bits represent decimal integers using base-2 representations.

Binary to Decimal

$(101)_2$ = 2^2 + 2^0 = 5

$(1010)_2$ = 2^3 + 2^1 = 10

Decimal to Binary: mod 2 and divide by 2

5 % 2 = 1 (5/2 = **2**)

2 % 2 = 0 (2/2 = **1**)

1 % 2 = 1 (1/2 = 0)

Base-3

$(201)_3 = 2 * 3^2 + 3^0 = 19$

Base-5

$(123)_5 = 1 * 5^2 + 2 * 5^1 + 3 * 5^0 = 38$

Base-16 (Hex)

$(A)_{16} = 10$

$(FF)_{16} = 255$

Converting between arbitrary bases? Just replace binary's 2 by other values and repeat the same conversion procedure.

- Using 4 bits, we can represent integers from $(0000)_2$ to $(1111)_2$
- Using N bits, we can represent $[0, 2^N)$
  - All non-negative
- To represent negative integers, we use the highest bit as the sign bit. Then using N bits, we can represent $[-2^{N-1}, 2^{N-1})$
  - 32-bit signed int max: 2,147,483,647 ~= $2 * 10^9$

ASCII

- A character is one byte or 8 bits.
- Each character in a string has a corresponding integer value.
  - ‘a’ = 97
  - ‘b’ = ‘a’ + 1 = 98

- This comes in handy when you’re processing characters (e.g. in a cipher implementation problem).
- This allows us to compare strings.

Lexicographical Ordering

- Default sorting order of strings.
- Letter order: a < b < c < …

How to:

- Compare char's one by one. If one char differs, then the string with the smaller char is smaller.
- If this goes beyond the length of one string (one string is a prefix of another), then the shorter string is smaller.

Examples:

"aaa" < "aab"
"bc" < "d"
"aa" < "aaa"
"" < "a"

Lexicographical Ordering (cont'd)

- Be extra careful when the string has non-letter characters!
  - "A.in" < "A-large.in" => false
- Be aware that uppercase letters are smaller than lowercase letters.
  - 'A' < 'a' => true

- We can extend this definitions to compare arrays.

[1, 1, 1] < [1, 1, 2]
[1, 4] < [2]
[1, 1] < [1, 1, 1]
[] < [1]

- Result exceeds the max/min value that a variable can store.

- For every problem:
  - Always read the constraints and determine the extremums for input variables, and the answer.
  - Determine what variable you need to use.
    - e.g. if you answer will exceed $2*10^9$, then you may consider using 64-bit.
    - Sometimes 64-bit's are not even enough. You may want big integers.

- Represent a larger number by an array of smaller numbers.
  - Like decimal representation: [1, 2, 3, 4, 5, 6, 7, 8, 9, 0] = 1234567890
  - However typically we use base-2 numbers.
- Using big integers you can represent virtually any large numbers.
- Java ships with BigInteger class, but C++ coders have to use their own.
  - * Python's integers extend automatically to big integers.

- However, modern APS tends to avoid big integers, because:
  - It's unfair for C++ competitors than Java
    - No longer a main reason. WF has supported Python, but not every problem is guaranteed to be solvable in Python. It is up to the contestant to master every language and decide what to use.
  - They are not much about being algorithmic.

- APS's way of creating fun problems without the need to worry about overflows.

Basics:

(a + b) % P = (a % P + b % P) % P
(a * b) % P = (a % P) * (b % P) % P

So now any intermediate value and the final result is in the range [0, P)

The lucky number: P = 1,000,000,007 (10^9 + 7)

- A large prime number
- Fits in 32-bit.
- Adding two integers below P still fits in 32-bit
  - a = b = 1,000,000,006
  - a + b = 2,000,000,012 < 2.1 * 10^9
- Multiplying two will overflow, but you can cast to 64-bit and then mod
  - int result = (long long) a * b % P
  - ~~long long result = a * b % P~~ Wrong!

Given N, what are the last 4 digits of “N!!” ? (double factorial) quotes for clarity)

Note that N!!, like N!, is a fast growing function. Assume O(N) can AC.

```
bowen@MacBook ~ $ python
Python 2.7.13 (default, Dec 17 2016, 23:03:43)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def double_factorial(n):
...     if n <= 1:
...         return 1
...     return n * double_factorial(n - 2)
...
>>> double_factorial(10)
3840
>>> double_factorial(100)
34243224702511976248246432895208185975118675053719198827915654463488000000000000L
>>> double_factorial(1000)
39939844265475088613053909446326587694914656127329508841493522644636696537839187866738558479315862976487247301007126872332878676700332343471477213721600529704688522774353670850
64790343283345982785772859940510918528070095138370429504769382461652184670792668564749747182728979673007824853940403174783379461981377594688094308995249697607727579137178606555
06921719422236343977822420657843146654419615078427126556112308853802759303355355152940145785364953577542572931070103507568654736612435505066087915531628866697355374514852059325
78756017485326071632607809593946114571618260053900221635003845177566094647495679952511113982165073735801248151774006017225908931505865903591160089887498517616931906892282380086
07755225687337923116872708482175000550889761043932312692198817965575874482087355854957667093854984963161313376186755900436597903368883575958968256738241094038338021793478788394
59285452597646011313895389346020262146141246676740687375355443330475516902518915052731339195969269104845657539480145492619768881861186330087442235891570050468250483068061790149
48154221077535522769055630188876097706645528590086960578527795414841862920355098616252996625851908253286400000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000L
>>> double_factorial(333)
12271230696851797411157742648219344131236778971527213508885252039879052290798654472102248992506280285365309098262518692374359028288201650246852159062778404615645465235480034159
63912460169407584858644585575213406000363594031363284053210681714279478220244248935533568461610644937407697197341729688136241532651838254119694406654161866754293441772460937 5L
>>>
```

Solution: Keep mod'ing!

```
1 int lastDigits(int n) {
2   int ans = 1;
3   for (int i = n; i >= 1; i -= 2) {
4     ans = ans * i % 10000;
5   }
6   return ans;
7 }
```

What if the last 6 digits?

```
1 int lastDigits(int n) {
2   int ans = 1;
3   for (int i = n; i >= 1; i -= 2) {
4     ans = ans * i % 1000000;
5   }
6   return ans;
7 }
```

Solution: Keep mod'ing!

```
1 int lastDigits(int n) {
2   int ans = 1;
3   for (int i = n; i >= 1; i -= 2) {
4     ans = ans * i % 10000;
5   }
6   return ans;
7 }
```

What if the last 6 digits?

```
1 int lastDigits(int n) {
2   int ans = 1;
3   for (int i = n; i >= 1; i -= 2) {
4     ans = ans * i % 1000000;
5   }
6   return ans;
7 }
```

lastDigits(333) => 609375 OK
lastDigits(3333) => -640625 What happened?

- **HW2**
  - Straightforward implementation tasks that help check your basic programming skill
    - Try to be bug free and achieve 1A!
  - Not much concern on efficiency (maybe a little?)
  - Base and modular arithmetic

Basic Programming Skill Checklist (HW2)

- **input:** read numbers and strings, read lines
- **output:** format numbers and strings, print floating point with rounding
- **variable types:** char, boolean, int, float, long, double
- **strings:** tokenization, reverse, get char, manipulate as array of chars
- **arrays:** 1D, 2D, 3D...
- **objects:** class/struct, attributes and methods
- **sorting:** numbers and strings, pairs, custom objects
- **assignment**: by copy or by reference, C++ pointers
- **memory:** alloc and dealloc
- **data structures and standard library**: usage of array/vector, linked list, (tree)set, (tree)map, hash table

Overall, you shall feel comfortable with and are able to handle any type of straightforward implementation tasks.