What is this course about?

- solve programming contest (real world?) type of problems;
- solve problems in a practical way:
  - do "just enough" work;
  - do it quickly;
  - do it correctly;
- learn how to classify problems and use appropriate existing algorithms in different situations.

Programming contest vs. "Real World"

- Specification of problem is usually very clear, although you have to make inferences using computer science or mathematical knowledge.
- Problem sizes are usually specified: you pick the right hammer for the job, not necessarily the algorithm with the best complexity.
- You do not want to overcomplicate your solution if it is good enough.
- Even one wrong case is completely wrong: forces you to think about all boundary and special cases, and not to accept programs with any bugs.
- Coding practices (e.g. commenting) may not matter, but will usually help get the job done faster and more accurately.

Topics

- Data structures
- Brute force/exhaustive search, backtracking
- Divide and conquer, greedy
- Dynamic programming
- Graphs
- Combinatorics
- Number theory
- Geometry
- String processing

Reading Input

- To read input into variables of different types, use `>>` operator as much as possible.
- Recall that `>>` skips all whitespaces, read as much characters as it can for the variable type, and leaves the "cursor" there.
- For strings, it will only read until the next whitespace.
- You can check the result of this operator to determine if the operation was successful. This can be used as an EOF test for us (though not in general).
- Recall: you can never predict EOF. You can detect it after you tried to read and failed.

Line-oriented Input

- Sometimes you want to process one line at a time.
- Use `getline` to read a line into a string. Return value indicates whether it is successful.
- If necessary, you can parse the string yourself, or you can use an `istringstream`.
- To "tokenize" a string, you may:
  - use `string::find` to look for the separator, extract the substring, and repeat.
  - replace all separators by whitespace (and whitespace to nonseparators), use `istringstream`.

Output

- Output is usually easier. Be precise about extra spaces and blank lines ("before", "after", "between", "separating").
- Use `setw()` manipulator for fixed-width fields.
- Use `setfill()` manipulator to fill the fields with specific characters other than space.
- Use `fixed` and `setprecision()` manipulators for printing floating-point numbers with a specified number of decimal places (careful about the rounding of "0.5").

Complexity

- From courses on algorithms, we know about the big-O notation (e.g. $O\left(n^2\right)$, $O\left(n \log n\right)$, etc.)
- They are used as a guideline for choosing an algorithm, but not an exact measure (unknown hidden constants).
- If a "slower" but simpler algorithm algorithm is good enough, use the slower one!
- $O(n!)$ is usually considered to be very bad, but for $n \leq 11$ it is just fine!
- If the number of iterations is less than 10 million per case, it should be okay (subject to change).

Reading the Problem

- Most problems are phrased in some "real world" setting.
- Note upper and lower limits on all quantities.
- Note all special properties: are duplicates allowed? Connected vs. unconnected graphs, etc.
- The right approach depends on these limits and special properties.
- Sometimes you can "guess" the complexity of the required approach by looking at these limits (e.g. small usually means brute force).
- Be careful about which data type (e.g. `int` vs. `long long`) you use.
- Also be aware of the amount of space (total and stack) that you are using.

Coding

- If you think you have the right algorithm (correctness and complexity), then you should code it soon...
- But double-check with the sample input/output, and/or some simple and special cases. There is no point coding an idea that will not work.
- Resist the temptation to code something that works most of the time, and then try to patch it until it works for "all cases".
- Use subroutines to help with code organization, especially the more complicated ones. Subroutines and variables with reasonable names can eliminate need of comments.

Coding

- Try to declare a variable only when you need it, and initialize at the same time (or as soon as possible). Use new blocks or routines to limit the scope.
- Using constructor/destructor to initialize and clear a data structure may be slightly less efficient compared to explicitly clearing it yourself, but you won't forget to do it.

Compiling and Testing

- Always turn on all compiler warnings: it may warn you about syntactically correct constructs that are wrong.
- Use I/O redirection to automate, and `diff` to check.
- Use `cat -vet` to see all blank spaces.
- Test on boundaries (both large and small, if practical).
- Test special cases.
- Reorder cases in input to catch uninitialized variables.
- Thinking about the special cases and boundaries before coding often give you more correct code/algorithm to begin with.

Debugging

- Print contents of variables at key points and carefully read the output while tracing.
- Print the code and simulate on paper.
- Try really small/simple input and see if your code/algorithm still breaks down.
- Optional: learn to use a debugger.