CS 491 CAP
Intro to Competitive Algorithmic Programming

Lecture 2
Data Structures & Libraries

Victor Gao

University of Illinois at Urbana-Champaign

September 8, 2017

Updates

- ICPC tryouts start Sept. 23rd!
  - Sign-up information will be sent to the mailing list next week
- Start practicing for the tryouts now!

Outline

- Why built-in data structures and libraries?
- Built-in data structures
  - Lists, stacks, queues, dictionaries, trees/heaps
  - Common operations
- Built-in libraries & utilities
  - Mathematics
  - Arbitrary-sized numbers
  - Character & string operations
  - Regular expressions

A note about Python

- Python is now allowed in the ICPC Regionals and World Finals
- Feel free to use python for the assignments in this class if the online judge supports it (some doesn't, like POJ)
- **However, make sure you know what is good and what is bad about python when you use it in the context this class (and competitive programming in general)**

A note about Python

Reasons to use Python:

- Faster implementation for trivial and easy problems where complexity is not an issue
- Native support for unbounded integer and complex list operations

Reasons not to use Python:

- All course assistants/instructors have familiarity primarily with C++/Java, so our ability to help with Python will be limited
- Python is not statically-typed, so it is easier to make mistakes and harder to debug
- MUCH slower than C++ or Java – **time limit exceeded** more likely, even for correct complexity solution
- Problems are not guaranteed to be solvable in Python (no Judges' solutions will be written in Python)
- Fundamental differences in primitives and libraries (e.g., lack of sorted ADTs)

Why built-in data structures and libraries are important?

Why Built-in DS and libraries?

- Using built-in data structures and libraries will greatly increase your coding speed
- Being familiar with the built-in's allows you to omit the details and think about the problems in a more abstract way
  - This ability is increasingly important as you start to deal with harder and harder problems
- The built-in's are tested over time and are very reliable
  - They are extremely unlikely to contain bugs that matter

Built-in Data Structures

Linear DS

- Standard arrays (C++/Java) or `array` module (Python)
  - Fixed size
  - Useful when you know the max size and can store it entirely in memory
  - Pros: Easy to index and manipulate (Java)
  - Pros: More compact than lists, but only usable for basic values (Python)
- `std::vector` (C++), `ArrayList` (Java), and `list` (Python)
  - Resizable
  - Random-access (i.e., can access by index)
  - Useful when you need to store variable number of items

Other linear DS

- **Linked List ADT:** `std::list` (C++) and `LinkedList` (Java)
  - Not random-access
  - No native Python library (but it's not necessary)
  - **Not normally used** – better to just use `std::vector` or `ArrayList`
- **Stack ADT:** `std::stack` (C++), `Stack` (Java), and `list` (Python)
  - Used for recursion, postfix, searching, bracket matching, etc.
- **Queue ADT:** `std::queue` (C++), `Queue` (Java), and `deque` (Python)
  - Used for searching, topological sort, etc.
- **Doubly-ended Queue ADT:** `std::deque` (C++), `ArrayDeque` (Java), and `deque` (Python)
  - Also known as deque ADT (pronounced "deck")
  - Used for "sliding window" problems
- **Priority Queue ADT:** `std::priority_queue` (C++), `PriorityQueue` (Java), and `list` + `heapq` module (Python)
  - Can be used when heap is required; always remains sorted
  - Used for simulation problems, Dijkstra's algorithm, Prim's algorithm

Common operations: Search

- Linear search:
  - Iterate through DS to find element index; **O(n)**
  - Implemented by `std::search` in `<algorithm>` (C++), but you should probably roll your own in all languages
- Binary search:
  - Keep dividing the DS in 2 and ignoring the half that doesn't contain the item; **O(log n)**
  - *Requires that the DS be sorted!*
  - Implemented by:
    - `std::lower_bound` and `std::bsearch` in `<algorithm>` (C++)
    - `Arrays.binarySearch` or `Collections.binarySearch` in Java
    - `bisect.bisect` in Python (better to sort your own list first, then use `bisect` for search than to use `bisect.insort`)

Common operations: Sort

- Many algorithms exist for sorting:
  - **O(n²)** algorithms: bubblesort, insertion sort, selection sort
  - **O(n log n)** algorithms: merge sort, quick sort, heap sort
  - Special purpose: radix sort, bucket sort
- First two classes require that items be comparable
  - E.g., numbers, strings, etc.
  - In C++, must overload the **<** (less than) operator
  - In Java, must implement **`Comparable`** interface
  - In Python, must implement the **`__eq__`** and **`__lt__`** methods
- Take an algorithms course if you are interested

Common operations: Sort

- In ICPC, we don't care about the algorithm, just the complexity (should be **O(n log n)** or less)
- You should almost ***never*** implement your own sort!
- Use `std::sort` in `<algorithm>` (C++), `Arrays.sort` or `Collections.sort` (Java), or built-in function `sorted()` or `list.sort` (Python)
  - If you need to write a custom sort, write a custom comparator [C++/Java] (see previous slide) or pass a key function to `sorted()` [Python] instead

Common operations (Sort)

- Special sorting algorithms in <algorithm> (C++):
  - `partial_sort`: implementation of heap sort; sorts only top *k* elements
  - `stable_sort`: ensures that elements of the same value stay in the same order as in the input
    - Java `Arrays.sort` and `Collections.sort` and Python `sorted()` and `list.sort` are stable sorts

Dictionaries

- Also known as table or map
- Manipulate key-value pairs
- Types:
  - Sorted (can binary search over elements)
    - Insert: **O(log n)**
    - Lookup: **O(log n)**
    - Delete: **O(log n)**
  - Unsorted
    - Insert: **O(1)***
    - Lookup: **O(1)***
    - Delete: **O(1)***

Dictionaries

- Unsorted dictionaries (also called **hash tables**):
  - `std::unordered_map` (C++11), `HashMap` (Java), and `dict` (Python)
- Sorted dictionaries:
  - `std::map` (C++), `TreeMap` (Java), and `dict` + `sorted()` (Python)

- Special-purpose dictionaries:
  - `LinkedHashMap` in Java and `collections.OrderedDict` (Python): traversal in order of insertion; **O(1)** lookup
  - `std::multimap` (sorted) and `std::unordered_multimap` (unsorted, C++11 only) in C++
    - Implements bag/multimap ADT
    - Supports multiple values for same key
    - Can achieve similar functionality with `HashMap<ArrayList>` in Java and `collections.defaultdict(list)` in Python

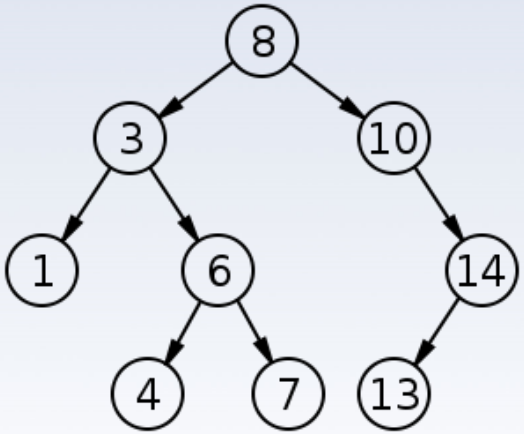
Sets

- Mathematical set ADT (element either exists in set or doesn't)
- Backing data structure behind dictionary keys
- Unsorted sets:
  - `std::unordered_set` (C++11), `HashSet` (Java), and `set` (Python)
- Sorted sets:
  - `std::set` (C++), `TreeSet` (Java), and `set` + `sorted()` (Python)

- Special-purpose sets:
  - `LinkedHashSet` in Java: traversal in order of insertion; **O(1)** operations

Binary search trees

- Sorted tree that can be binary searched
  - Requirement: **must be balanced** for **O(log n)** operations!
- No general tree data structure in any of the languages – roll your own
- However, you can use a sorted dictionary or set to get BST functionality



Other data structures

- Bit sets and bit masks
- Disjoint-set
- Graphs
- Segment tree

- Will cover in later classes or CS 491 WF

Questions so far?

Built-in Libraries & Utilities

Mathematics

- Most basic math operations are built-in

- Topics:
  - Basic functions (including trigonometric, rounding, exponentiation, etc.)
  - Integer base conversion
  - Arbitrary precision numbers

Basic functions

- Available in `<cmath>` (C++), `java.lang.Math` (Java), or `math` module (Python)
- Functions:
  - sin, cos, tan, asin, acos, atan, atan2
    - And their hyperbolic equivalents
  - abs (and fabs in C++/Python), copysign (C++11/Python) and signum (Java)
  - ceil, floor, round (C++/Java), trunc (Python)
  - max, min
  - sqrt, pow, hypot, log, log10, log1p, exp, expm1
- Constants:
  - e (call `exp(1)` in C++), π (call `acos(-1)` in C++)
- Complex numbers: `<complex>` (C++) and `cmath` module (Python)

Conversion between types

- Number to string
  - C++11 (in `<string>`): `std::to_string`
  - C++ (in `<stdlib.h>`, but not supported by all compilers): `itoa`
  - Java (in `java.lang.String`): `String.valueOf`
  - Python (built-in function): `str()`
- String to number
  - C++ (in `<cstdlib>`): `stoi`, `stol`, `stoll`, `stof`, `stod`, `stold`, etc.
  - Java (in `java.lang.Number`): `Integer.parseInt`, `Long.parseLong`, `Float.parseFloat`, `Double.parseDouble`, etc.
  - Python (built-in functions): `int()`, `float()`, etc.
- For string formatting, can also use `stringstream` and `sprintf` in C++, `String.format` in Java, and built-in function `format()` in Python

Integer base conversion

- Specify base in number-to-string and string-to-number functions
  - In C++, use `itoa` and `stoi`/`stol`/`stoll`
  - In Java, use (`Integer`/`Long`)`.toString`/(`parseInt` or `parseLong`)
  - In Python, use `int()` and `format()`
    - Only binary (`'b'`), octal (`'o'`), decimal (`'d'`), and hexadecimal (`'x'`) are supported by `format()`; for all else, roll your own
  - Look at API reference and *practice on your own*!

Arbitrary precision numbers

- Normal numeric types have fixed limits (e.g., 32-bit or 64-bit integer/float)
- We want types that can support arbitrary-length numbers

- Python: `int` and `decimal`
  - **Note:** Python `decimal` and Java `BigDecimal` differ in functionality
- Java: `BigInteger` and `BigDecimal`
  - Support addition, subtraction, multiplication, division*, exponentiation, negation
  - BigInteger also supports bitwise operations
  - Read API for more details – lots of good stuff there!
- Not available in standard C++ (there are external libraries, but can't use them in ICPC)

Character operations

- Checking to see if a character is of a certain class (uppercase, lowercase, punctuation, numeric, etc.)
- In C++, list of functions in **`std::<cctype>`**
- In Java, static methods in **`java.lang.Character`**
- In Python, built-in string methods (can be applied without importing anything)

String operations

- Can use strings in clever ways to make your life easier
- `StringBuilder` in Java, `std::stringstream` in C++, and list comprehensions in Python to construct strings dynamically
  - Can also append to regular `std::string` in C++; C++ strings are mutable
- Reverse a string: `std::reverse` (C++ in `<algorithm>`), `StringBuilder.reverse` (Java), and `[::-1]` (extended slicing) (Python)

- Example: replace all 'a's in a string with 'b's, and vice versa
- Wrong: `str.replace('a', 'b').replace('b', 'a')`
- Right: `str.replace('a', '^').replace('b', 'a').replace('^', 'b')`

Regular expressions

- Regular expressions allow you to search for or match strings based on patterns instead of actual characters
  - For example, to check if a string is made up of only letters, can match on `r"[a-zA-Z]*"`
- Very powerful for string parsing and sanitizing
- `<regex>` library (C++11), `java.util.regex` (Java), and `re` module (Python)
  - To specify pattern, use `basic_regex` (C++11) or `Pattern` (Java)
  - To find matches, use `match_results` (C++11) or `Matcher` (Java)
  - In Python, can do both using module functions `re.search`, `re.match`, `re.findall`, etc.

- Read API reference for your language for more details
- Use an online tutorial like Java's tutorial to learn regex
  - https://docs.oracle.com/javase/tutorial/essential/regex/

Miscellaneous

- Random numbers (rarely useful in ICPC):
  - `std::rand` (C++ in `<cstdlib>`), `java.util.Random` (Java), and `random` module (Python)
- Permutations:
  - `std::next_permutation` and `std::prev_permutation` (C++)
  - `itertools` module in Python provides useful tools, but is different from above C++ functions
  - Roll your own for Java, but you can find example code online
- Read the API reference for all of the classes and functions covered today (also review `<algorithm>` for C++ users)!

Python-specific miscellaneous

- Learn how to use lambda expressions, generators, and list comprehsions
  - Shorter to code and more efficient within Python
- Learn about `collections` module
  - `namedtuple`, `Counter`, and `defaultdict` are all very useful for ad hoc problems

Questions?

Resources for this lecture

- Steven Halim's Competitive Programming book
  - Link on Syllabus page
- C++ Reference – http://en.cppreference.com/w/
- Java 8 API – http://docs.oracle.com/javase/8/docs/api/
- Python Documentation
  - Python 2: https://docs.python.org/2/index.html
  - Python 3: https://docs.python.org/3/index.html