

Competitive Programming

Frank Takes

LIACS, Leiden University

<https://liacs.leidenuniv.nl/~takesfw/CP>

Lecture 2 — Data structures and libraries

Recap

After last week . . .

- You are familiar with the basic concepts of competitive programming,
- have the practical individual skills in C or C++ to participate in a programming contest,
- know roughly how input size constraints dictate maximum time complexity,
- can use DOMjudge and have an account at <https://domjudge.liacs.nl>,
- know the specifics of program input/output via `stdin/stdout`
- understand how to output the various data types,
- and have seen a dynamic input problem.

Practicalities

- C++ 11 (compiled using `g++ -g -O2 -std=gnu++11`)
- JAVA 8
- Character encoding: UTF-8
- Clarification requests “work”

Your programming language(s)

- 1 C, C++ and Java
- 2 C and Java
- 3 C++ and Java
- 4 C and C++
- 5 C
- 6 C++
- 7 Java

Data structures



Linear data structures

Standard Template Library

- C++ Standard Library
- Data structures
e.g., vector, list, queue, stack, set, map
- Functions in `#include<algorithm>` (but also `bitset` and `string`),
e.g., `sort`, `min`, `max`, `count`, `binary_search`
- Generally fast enough
- (Re-)familiarize yourself:
<http://www.cplusplus.com/reference/stl>

Linear data structures

- Static: arrays `int A[100];`
Or `array<int, 3> A;`
- Dynamic: C++ STL vector `vector<int> A(100);`
Alternative for `vector<bool> B` `bitset<100> A;`
- Linked lists (e.g., using `list<int> A`) are rarely used

Linear data structures

- Static: arrays `int A[100];`
Or `array<int, 3> A;`
- Dynamic: C++ STL vector `vector<int> A(100);`
Alternative for `vector<bool> B` `bitset<100> A;`
- Linked lists (e.g., using `list<int> A`) are rarely used
- Common operations:
 - Sort
 - Search

Problem type: sorting

- Given some linear data structure, sort it
- Complexity: $O(n \log n)$
- $O(n)$ for special cases (e.g., small number of distinct elements)
- In practice: `algorithm::sort`
 - Works for basic data types (int, double etc.)

```
vector<int> myvalues;  
...  
sort(myvalues.begin(), myvalues.end());
```
 - Custom data types if the comparison operator `<` is defined

Sorting with custom data types

```
// a coin is worth its value, but worth slightly more when unused
struct Coin {
    int value;
    bool used;
}; // Coin

bool operator < (const Coin & a, const Coin & b) {
    if(a.value == b.value)
        return(a.used > b.used);
    return a.value < b.value;
} // < operator for Coin

...
vector<Coin> coins;
... // fill it with data
sort(coins.begin(), coins.end());
...
```

Problem type: search

- Given some linear data structure, search for an element
- Search for an element once or less than $\log(n)$ times: linear scan
- Searching more often:
 - 1 sort the list in $O(n \log n)$
 - 2 use binary search to find elements in $O(\log n)$
e.g., from `algorithm::binary_search`
or the easier `algorithm::lower_bound`

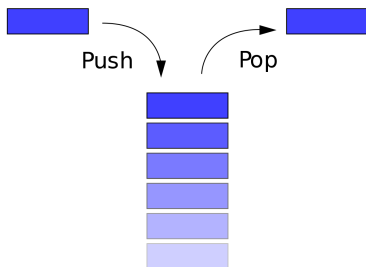
Stack

■ Stack

Operations: `push()`, `top()`, `pop()`, etc.

(STL implementation is very similar to `vector`)

```
stack<int> S;
```

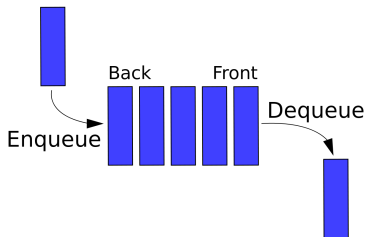


Queue

■ Queue

Operations: `front()`, `push()`, `pop()`, etc.

```
queue<int> Q;
```



■ Double-ended queue

Operations: `front()`, `back()`, `push_back()`, `pop()`, etc.

```
deque<int> Q;
```

Nonlinear data structures

Priority queue

```
// a coin is worth its value, but worth slightly more when unused
struct Coin {
    int value;
    bool used;
}; // Coin

struct CustomCompare {
    bool operator() (const Coin & a, const Coin & b) {
        if(a.value == b.value)
            return(a.used > b.used);
        return a.value < b.value;
    } // < operator for myCoin
}; // CustomCompare

...
priority_queue<int, std::vector<int>, CustomCompare> Q;
// default is std::less<int>, sometimes std::greater<int> suffices
...
```

Mapping and hashing

- Given some key, check if it exists
- Given some key, find a bit of data

```
set<keytype> S;  
map<keytype, datatype> M;
```



Sets

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

int main() {
    int t;
    set<int> S;
    int N = 100;

    for(int i=0; i<N; i++)
        S.insert(rand() % 100);

    cout << S.size() << endl;
    // outputs 68 (of course, this is dependent on random seed),
    // value of 100 expected as N gets larger

    return 0;
} // main
```



Map

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

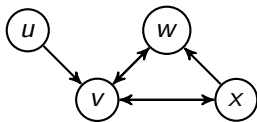
int main() {
    int t;
    map<string, int> M;
    M["een"] = 1;
    M["dertien"] = 13;

    cout << M["een"] + M["dertien"] << endl;
    // the above outputs: 14

    return 0;
} // main
```

Graphs

- Directed graph $G = (V, E)$ with $E \subseteq V \times V$
- Variants: undirected (simple) graph, bipartite graph, weighted graph, directed acyclic graph (dag), tree
- Example below has $V = \{u, v, w, x\}$ and $E = \{(u, v), (v, w), (v, x), (w, v), (x, v), (x, w)\}$ so, $|V| = 4$ and $|E| = 6$



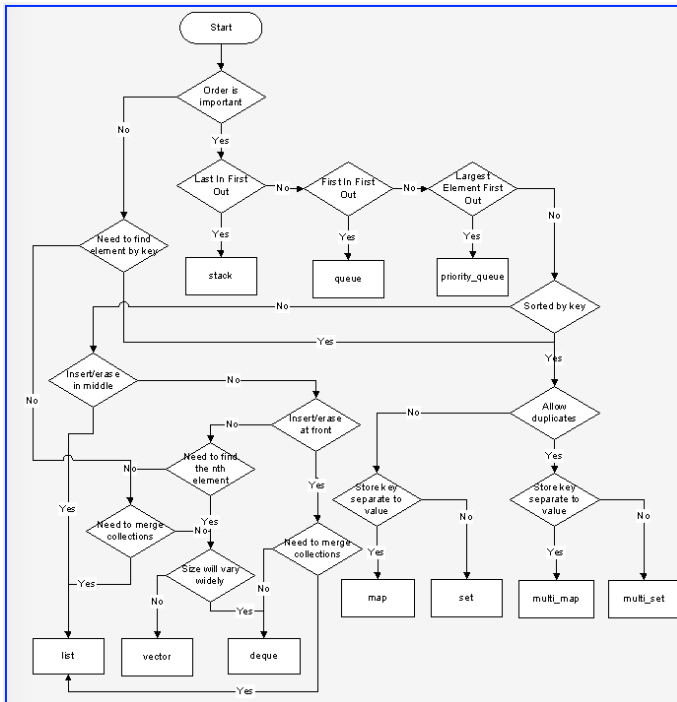
Graph datastructures

- **Adjacency matrix:** store whether all $N \times N$ node pairs are linked
`bool A[N][N];`
`int` for weights, additional instances for edge labels
- **Adjacency list:** store the neighbors of all N nodes
`vector< vector<int> > A(N);`
store something more complex than `int` for edge labels or weights
- **Edge list:** store all M existing edges linearly
`int source[M]; int target[M];`
additional arrays for edge labels or weights

Here, N is the number of nodes and M the number of edges

Graph datastructures

	Adjacency list	Adjacency matrix	Edge list
Storage	$O(V + E)$	$O(V ^2)$	$O(E)$
Add vertex	$O(1)$	$O(V ^2)$	$O(1)$
Add edge	$O(1)$	$O(1)$	$O(1)$
Remove vertex	$O(E)$	$O(V ^2)$	$O(E)$
Remove edge	$O(E)$	$O(1)$	$O(E)$
Query $(u, v) \in E?$	$O(V)$	$O(1)$	$O(E)$



Headers	Sequence containers					Associative containers				Unordered associative containers				Container adaptors		
	<array>	<vector>	<deque>	<forward_list>	<list>	<set>	<multiset>	<map>	<multimap>	<unordered_set>	<unordered_multiset>	<unordered_map>	<unordered_multimap>	<stack>	<queue>	<priority_queue>
(constructors)	(mpfcl)	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
(destructors)	(mpfcl)	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
operator*	(mpfcl)	operator	operator	operator	operator	operator	operator	operator	operator	operator	operator	operator	operator	operator	operator	operator
assign		assign	assign	assign	assign											
begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin
cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin
end	end	end	end	end	end	end	end	end	end	end	end	end	end	end	end	end
cbend	cbend	cbend	cbend	cbend	cbend	cbend	cbend	cbend	cbend	cbend	cbend	cbend	cbend	cbend	cbend	cbend
rbegin	rbegin	rbegin	rbegin		rbegin	rbegin	rbegin	rbegin	rbegin							
crbegin	crbegin	crbegin	crbegin		crbegin	crbegin	crbegin	crbegin	crbegin							
rend	rend	rend	rend		rend	rend	rend	rend	rend							
crend	crend	crend	crend		crend	crend	crend	crend	crend							
at	at	at	at					at			at					
operator[]	operator[]	operator[]	operator[]					operator[]			operator[]					
front	front	front	front	front	front									front		top
back	back	back	back	back	back									top	back	top
empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty
size	size	size	size	size	size	size	size	size	size	size	size	size	size	size	size	size
max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size
resize	resize	resize	resize	resize	resize											
capacity	capacity	capacity	capacity	capacity	capacity											
reserve	reserve	reserve	reserve	reserve	reserve					reserve	reserve	reserve	reserve			
shrink_to_fit	shrink_to_fit	shrink_to_fit	shrink_to_fit	shrink_to_fit	shrink_to_fit					reserve	reserve	reserve	reserve			
clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear
insert	insert	insert	insert	insert_after	insert	insert	insert	insert	insert	insert	insert	insert	insert	insert	insert	insert
emplace	emplace	emplace	emplace	emplace_after	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace
emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint
erase	erase	erase	erase	erase_after	erase	erase	erase	erase	erase	erase	erase	erase	erase	erase	erase	erase
push_front	push_front	push_front	push_front	push_front	push_front											
emplace_front	emplace_front	emplace_front	emplace_front	emplace_front	emplace_front											
pop_front	pop_front	pop_front	pop_front	pop_front	pop_front											pop
push_back	push_back	push_back	push_back	push_back	push_back									push	push	push
emplace_back	emplace_back	emplace_back	emplace_back	emplace_back	emplace_back									emplace	emplace	emplace
pop_back	pop_back	pop_back	pop_back	pop_back	pop_back									pop	pop	pop
swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap
merge	merge	merge	merge	merge	merge											
splice	splice	splice	splice	splice	splice											
remove	remove	remove	remove	remove	remove											
remove_if	remove_if	remove_if	remove_if	remove_if	remove_if											
reverse	reverse	reverse	reverse	reverse	reverse											
unique	unique	unique	unique	unique	unique											
sort	sort	sort	sort	sort	sort											
count	count	count	count	count	count	count	count	count	count	count	count	count	count	count	count	count
find	find	find	find	find	find	find	find	find	find	find	find	find	find	find	find	find
lower_bound	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound
upper_bound	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound
equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range
key_comp	key_comp	key_comp	key_comp	key_comp	key_comp	key_comp	key_comp	key_comp	key_comp	key_comp	key_comp	key_comp	key_comp	key_comp	key_comp	key_comp
value_comp	value_comp	value_comp	value_comp	value_comp	value_comp	value_comp	value_comp	value_comp	value_comp	value_comp	value_comp	value_comp	value_comp	value_comp	value_comp	value_comp
hash_function	hash_function	hash_function	hash_function	hash_function	hash_function	hash_function	hash_function	hash_function	hash_function	hash_function	hash_function	hash_function	hash_function	hash_function	hash_function	hash_function
key_eq	key_eq	key_eq	key_eq	key_eq	key_eq	key_eq	key_eq	key_eq	key_eq	key_eq	key_eq	key_eq	key_eq	key_eq	key_eq	key_eq
Allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator

Problem types

- 1 Simulation
- 2 Brute-force
- 3 Searching
- 4 Sorting
- 5 Graphs
- 6 Greedy
- 7 Divide and conquer
- 8 Dynamic programming
- 9 String processing
- 10 Geometry
- 11 Mathematics

Course organization (continued)

Throughout the course

- Practice

Throughout the course

- Practice
- Practice
- Practice
- Practice
- Practice
- Practice

Concrete course format

- Contest (“soft”) individual, 20%
starts February 27, 2020
with short report deadline March 9, 2020
- Presentation and report individual, 35%
throughout semester; list of topics divided in week 3
deadline for topic report: May 4, 2020
- Three (live) programming contests teams, $3 \times 15 = 45\%$
In March and April 2020 (dates TBD)

Planning live contests

- Three 3-4 hour contests
- Problemset of 6–9 problems
- One computer
- Teams of 2–3 students
- Only use programming language manuals
- Team manual is allowed
- So, when? ...

- Kattis
- Online practice platform
- `https://open.kattis.com`
- We will use it for weekly exercises
- Create an account
- Will not be used for course contests or grading

Lab session & “Homework”

- Try out Java (if you want)
- (Re)-familiarize yourself with different algorithms and data structures via <https://github.com/gibsjose/cpp-cheat-sheet/blob/master/Data%20Structures%20and%20Algorithms.md>
- (Re)-familiarize yourself with the C++ standard library via <http://www.cplusplus.com/reference/stl>
- Get started with the practice problems of week 2: <https://open.kattis.com/contests/yhoxxs>

This course, in particular these slides, are largely based on:

- Antti Laaksonen, *Guide to Competitive Programming*, Springer, 2017.
- Steven Halim and Felix Halim, *Competitive Programming 3*, Lulu.com, 2013.
- T-414-AFLV: A Competitive Programming Course, <https://github.com/SuprDewd/T-414-AFLV>

Where applicable, full credit for text, images, examples, etc. goes to the authors of these books.