

# Complexity and standard libraries

---

Atli FF & Arnar Bjarni Arnarson

August 18, 2025

School of Computer Science

Reykjavík University

# Overview for today

- Comments on inputs
- Basic data types
- Complexities
- Standard libraries
- Why we need data structures
- Standard library data structures

# Inputs

- As mentioned in the last lecture, when doing I/O **only print what is asked of in the output description.**
- Do not use `input("Please enter a number:")` or something equivalent, this prints to `stdout`.

Language	Stdin	Stdout
C	<code>scanf</code>	<code>printf</code>
C++	<code>cin</code>	<code>cout</code>
Python	<code>input()</code>	<code>print()</code>
Java	<code>Scanner(System.in)</code>	<code>System.out.print</code>

# Speeding up I/O

- If fast I/O is needed, `sys.stdin` and `sys.stdout` are slightly faster in python.
- Similarly for fast I/O in Java, look up Kattio, it is much *much* faster. It can be found on github.
- To speed up C++ I/O somewhat one can use `ios_base::sync_with_stdio(false)`. Note that if this is done `scanf` or `printf` no longer sync up with `cin` and `cout`, so you have to pick one set and stick to it. Similarly `cin.tie(nullptr)` can speed things up as `cout` will no longer flush before `cin` is used, but this can cause problems when this behaviour is desired.

# Basic data types

---

# Basic data types

- Most languages contain some version of the following:
  - `bool`: a boolean (`true/false`)
  - `char/int8_t`: an 8-bit signed integer (often used to represent characters with ASCII)
  - `int32_t`, `int64_t`, ...: Signed fixed size integers
  - `uint32_t`, `uint64_t`, ...: Unsigned fixed size integers
  - `float`: an IEEE 32-bit floating-point number
  - `double`: an IEEE 64-bit floating-point number
  - `string`: a string of characters

# Basic data types

Type	Bytes	Min value	Max value
bool	1		
int8_t	1	-128	127
int16_t	2	-32768	32767
int32_t	4	-2148364748	2147483647
int64_t	8	-9223372036854775808	9223372036854775807
	$n$	$-2^{8n-1}$	$2^{8n-1} - 1$

Type	Bytes	Min value	Max value
uint8_t	1	0	255
uint16_t	2	0	65535
uint32_t	4	0	4294967295
uint64_t	8	0	18446744073709551615
	$n$	0	$2^{8n} - 1$

Type	Bytes	Min value	Max value	Precision
float	4	$\approx -3.4 \times 10^{38}$	$\approx 3.4 \times 10^{38}$	$\approx 7$ digits
double	8	$\approx -1.7 \times 10^{308}$	$\approx 1.7 \times 10^{308}$	$\approx 14$ digits

# Big integers

- What if we need to represent and do computations with very large integers, i.e. something that doesn't fit in a `__int128`
- Simple idea: Store the integer as a string
- But how do we perform arithmetic on a pair of strings?
- We can use the same algorithms as we learned in elementary school
  - Addition: Add digit-by-digit, and maintain the carry
  - Subtraction: Similar to addition
  - Multiplication: Long multiplication
  - Division: Long division
  - Modulo: Long division



## Example problem: Simple Addition

- <https://open.kattis.com/problems/simpleaddition>
- As can be seen on the statistics for this problem, the fastest are C/C++. But big integer operations have to be implemented manually in those languages. For these kinds of problems it can be easier to use a language that has built-in support, like Java, Python or Haskell.

# Time complexities

---

# What is a time complexity?

- Saying a program runs in  $\mathcal{O}(f(n))$  means that for some  $C, n_0$  the program will take at most  $Cf(n)$  steps to finish for  $n \geq n_0$
- Ignoring constants is necessary, otherwise you could change the time complexity just by making the CPU faster or adding more cores
- Time complexities are very useful for napkin math on whether a solution will pass time constraints
- For example  $\mathcal{O}(n^2)$  means that as  $n$  increases, the number of steps the program needs grows at most like  $n^2$ . So if we double  $n$ , the runtime might get multiplied by up to 4

# Calculate time complexities

- A good rule of thumb is that we have  $10^8$  operations per second

# Calculate time complexities

- A good rule of thumb is that we have  $10^8$  operations per second
- Say we want to sort  $n \leq 10^6$  integers in 3 seconds.
- Can we use a  $\mathcal{O}(n^2)$  bubble sort or do we need to implement the more complex  $\mathcal{O}(n \log(n))$  merge sort?

## Calculate time complexities

- A good rule of thumb is that we have  $10^8$  operations per second
- Say we want to sort  $n \leq 10^6$  integers in 3 seconds.
- Can we use a  $\mathcal{O}(n^2)$  bubble sort or do we need to implement the more complex  $\mathcal{O}(n \log(n))$  merge sort?
- Bubble sort would take  $\sim 10^{12}$  operations or about  $10^4$  seconds, which is far too slow.
- The merge sort would be around 0.2 seconds, which suffices.

## Time complexities cntd.

- Always use the simplest solution that suffices. If  $n$  had been  $10^3$  bubble sort would suffice.
- It can be good to be able to estimate these things quick in your head.
- Rules of thumb can be useful, things like  $2^{10} \approx 10^3$ .
- Logarithms are usually base 2, so like earlier if  $n = 10^6$  for  $n \log(n)$  we can estimate it as  $10^6 \log_2(2^{20})$  or  $2 \cdot 10^7$ .

# Complexity overview

$n$	Slowest Accepted Algorithm	Example
$\leq 10$	$O(n!)$	Enumerating a permutation
$\leq 15$	$O(2^n \times n^2)$	Traveling salesperson DP
$\leq 20$	$O(2^n), O(n^5)$	Bitmask DP
$\leq 50$	$O(n^4)$	Blossom algorithm
$\leq 10^2$	$O(n^3)$	Floyd Warshall algorithm
$\leq 10^3$	$O(n^2)$	Bubble/Selection/Insertion sort
$\leq 10^5$	$O(n \log_2 n)$	Merge sort, building a Segment tree
$\leq 10^6$	$O(n)$	Linear scans like prefix sums
$> 10^8$	$O(\log_2 n), O(1)$	Direct formulas or digit operations



## Standard libraries

---

# Language features

- Kattis allows the use of standard libraries, so get acquainted with what your language of choice has to offer.
- Kattis does not have other packages, like `algs4` (Java) or `boost` (C++)
- C++ sorts with `sort(a.begin(), a.end())`, python has `a.sort()` and Java has `Arrays.sort(a)`.
- All three languages support common mathematical operations like square roots and complex numbers.
- C++ can do binary search with `lower_bound` and `upper_bound`, python can `import bisect`.
- There's plenty more! Regex, pseudo-randomness and plenty of data structures.

# Language features

- The C++ standard library has many useful features like this
- `reverse` can reverse a vector or array in place
- `rotate` can rotate a vector or array in place
- `count` and `count_if` can count elements, or count elements satisfying a predicate function
- `find` and `find_if` can find an element, or find an element satisfying a predicate function
- And many more!

# Data structures

---

# Why do we need data structures?

- Sometimes our data needs to be organized in a way that allows one or more of
  - Efficient querying
  - Efficient inserting
  - Efficient deleting
  - Efficient updating
- Sometimes we need a better way to represent our data
  - How do we represent large integers?
  - How do we represent graphs?
  - How do we represent equivalence relations?
- Data structures help us achieve those things

# Data structures you should (hopefully) be familiar with

- Static arrays
- Dynamic arrays
- Linked lists
- Stacks
- Queues
- Priority queues
- Ordered trees
- Hash maps

# Data structures you should (hopefully) be familiar with

- Static arrays - `T arr[n]`
- Dynamic arrays - `vector<T>`
- Linked lists - `list<T>`
- Stacks - `stack<T>`
- Queues - `queue<T>`
- Priority queues - `priority_queue<T>`
- Ordered trees - `set<T>`
- Hash maps - `unordered_map<K,V>`

# Data structures in standard libraries

- Usually it's best to use the standard library implementations
  - Almost surely bug-free and fast
  - We don't need to write any code
- Sometimes we need our own implementation
  - When we want more flexibility
  - When we want to customize the data structure
- And sometimes we need data structures not in the standard library, but that waits until later in the course



# Applications of Arrays and Vectors

- Too many to list
- Most problems require storing data, usually in an array
- Vectors are similar but are dynamically allocated

Operation	Array complexity	Vector complexity
Access element	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Append element	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Delete/insert element	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Combine arrays/vectors	$\mathcal{O}(n)$	$\mathcal{O}(n)$

# Vector

- Capacity is the number of slots allocated for items
- Size is the number of slots in use
- Has some fixed capacity upon initialization
- Can append items until size reaches capacity
- Can remove from the back
- How to insert / erase at other positions?
- How to append when size equals capacity?

## Vector - insert/erase

- Want to insert  $x$  and index  $i$ .
- Before that we must move item at index  $i$  to index  $i + 1$ .
- Before that we must move item at index  $i + 1$  to index  $i + 2$ .
- ...
- So the first step is moving item at index  $n$  to index  $n + 1$ .
- Erasing works similarly, but in reverse.

## Vector - resizing

- Need to allocate a new larger block of memory
- Then copy from old to new
- Finally free up the old block of memory
- Now you can finish the append/insert operation
- Can also shrink the vector if size is much lower than capacity.
- Manual calls to `resize` / `reserve` allow for optimizations

# Applications of Linked lists

- Very rarely used
- Mostly used when concatenation of arrays or insertion outside of endpoints needs to be fast
- Nodes can be allocated in a batch to improve memory layout

Operation	Complexity
Access element	$\mathcal{O}(n)$
Append/prepend element	$\mathcal{O}(1)$
Delete/insert at pointer	$\mathcal{O}(1)$
Combine lists	$\mathcal{O}(1)$

# Applications of Stacks

- Processing events in a last-in first-out order
- Simulating recursion
- Depth-first search in a graph
- Matching brackets
- And a lot more

Operation	Complexity
Push element	$\mathcal{O}(1)$
Access most recent element	$\mathcal{O}(1)$
Pop most recent element	$\mathcal{O}(1)$

# Applications of Queues

- Processing events in a first-in first-out order
- Breadth-first search in a graph
- And a lot more

Operation	Complexity
Push element	$\mathcal{O}(1)$
Access oldest element	$\mathcal{O}(1)$
Pop oldest element	$\mathcal{O}(1)$

# Applications of Heaps/Priority Queues

- Processing events in order of priority
- Finding a shortest path in a graph
- Some greedy algorithms
- And a lot more

Operation	Complexity
Push element	$\mathcal{O}(\log(n))$
Access highest priority element	$\mathcal{O}(1)$
Pop highest priority element	$\mathcal{O}(\log(n))$



# Applications of Sets

- Keep track of distinct items
- If implemented as a binary search tree:
  - Find next greater element
  - Count how many elements are less than a given element
  - Find the  $k$ th largest element
- And a lot more

Operation	Complexity
Access/check for element	$\mathcal{O}(\log(n))$
Insert/delete element	$\mathcal{O}(\log(n))$

# Applications of Hash maps

- Associating a value with a key
- As a frequency table
- And a lot more

Operation	Complexity
Access/check for element	$\mathcal{O}(1)$ average, $\mathcal{O}(n)$ worst case
Insert/delete element	$\mathcal{O}(1)$ average, $\mathcal{O}(n)$ worst case

## Using data structures

---

# How to use data structures

- In many cases you don't need fancy data structure, you just have to use simple ones in a smart way.
- Let us consider next the problem of NGE (Next Greater Element).
- We have an array of numbers. For each of the numbers we want to know where the next element to the right that is greater than it is located.
- Example: [6, 2, 4, 7, 1] -> [3, 2, 3, NULL, NULL].  
Note that this is zero-indexed.

- We could always just walk to the right for every element.

- We could always just walk to the right for every element.
- But what happens for a decreasing list then?

- We could always just walk to the right for every element.
- But what happens for a decreasing list then?
- The time complexity is  $\mathcal{O}(n^2)$ ! Far too slow.

# Smart NGE

- Let us consider a better algorithm. Let us start with a stack  $s$ .
- We then walk through the list from left to right and add values to the stack as we go.
- If what we want to put on the stack is bigger than the top element, the top element must have our current element as its NGE.
- Thus we set that in our output and pop the top element, repeating as necessary, before putting our new element on top.
- At the end our output vector will contain the answer.
- This will then give us a  $\mathcal{O}(n)$  NGE solution. Let's see an example.



```
0 1 2 3 4 5 6 7  
[2 3 1 5 7 6 4 8]
```

|

```
0 1 2 3 4 5 6 7  
[x x x x x x x x]
```

h: []

```
0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
|
0 1 2 3 4 5 6 7
[x x x x x x x x]
h: []
```

```
  0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
|
  0 1 2 3 4 5 6 7
[x x x x x x x x]
h: [0]
```

```
0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
|
0 1 2 3 4 5 6 7
[x x x x x x x x]
h: [0]
```

```
0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
^ |
0 1 2 3 4 5 6 7
[x x x x x x x x]
h: [0]
```

```
0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
^ |
0 1 2 3 4 5 6 7
[1 x x x x x x x]
h: [0]
```

```
0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
|
0 1 2 3 4 5 6 7
[1 x x x x x x x]
h: []
```

```
0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
|
0 1 2 3 4 5 6 7
[1 x x x x x x x]
h: [1]
```



```
0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
|
0 1 2 3 4 5 6 7
[1 x x x x x x x]
h: [1]
```

```
0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
  ~ |
0 1 2 3 4 5 6 7
[1 x x x x x x x]
h: [1]
```

```
0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
|
0 1 2 3 4 5 6 7
[1 x x x x x x x]
h: [1]
```

```
0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
```

|

```
0 1 2 3 4 5 6 7
[1 x x x x x x x]
```

```
h: [1 2]
```

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	x	x	x	x	x	x	x]

h: [1 2]

```
0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
```

~ |

```
0 1 2 3 4 5 6 7
[1 x x x x x x x]
```

h: [1 2]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

~ |

0	1	2	3	4	5	6	7
[1	x	3	x	x	x	x	x]

h: [1 2]

```
0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
  ~   |
0 1 2 3 4 5 6 7
[1 x 3 x x x x x]
h: [1]
```



```
0 1 2 3 4 5 6 7
[2 3 1 5 7 6 4 8]
  ~   |
0 1 2 3 4 5 6 7
[1 3 3 x x x x x]
h: [1]
```

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	x	x	x	x	x]

h: []

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	x	x	x	x	x]

h: [3]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	x	x	x	x	x]

h: [3]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

^ |

0	1	2	3	4	5	6	7
[1	3	3	x	x	x	x	x]

h: [3]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

^ |

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	x	x]

h: [3]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	x	x]

h: []

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	x	x]

h: [4]



0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	x	x]

h: [4]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

~ |

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	x	x]

h: [4]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	x	x]

h: [4]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	x	x]

h: [4 5]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	x	x]

h: [4 5]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

~ |

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	x	x]

h: [4 5]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	x	x]

h: [4 5]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	x	x]

h: [4 5 6]



0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	x	x]

h: [4 5 6]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

~ |

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	x	x]

h: [4 5 6]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]
					^		

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	7	x]
h:	[4	5	6]				

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

~ |

0	1	2	3	4	5	6	7
[1	3	3	4	x	x	7	x]

h: [4 5]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

~ |

0	1	2	3	4	5	6	7
[1	3	3	4	x	7	7	x]

h: [4 5]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

~ |

0	1	2	3	4	5	6	7
[1	3	3	4	x	7	7	x]

h: [4]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

~ |

0	1	2	3	4	5	6	7
[1	3	3	4	7	7	7	x]

h: [4]

0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	4	7	7	7	x]

h: []



0	1	2	3	4	5	6	7
[2	3	1	5	7	6	4	8]

|

0	1	2	3	4	5	6	7
[1	3	3	4	7	7	7	x]

h: [8]

```
0 1 2 3 4 5 6 7
```

```
[2 3 1 5 7 6 4 8]
```

```
0 1 2 3 4 5 6 7
```

```
[1 3 3 4 7 7 7 x]
```

```
h: [8]
```

<MINTED>

# Augmenting Data Structures

- Sometimes we can store extra information in our data structures to gain more functionality
- Usually we can't do this to data structures in the standard library (but there are exceptions, `gnu_pbds`)
- Need our own implementation that we can customize
- Sometimes this functionality is simply better time complexity

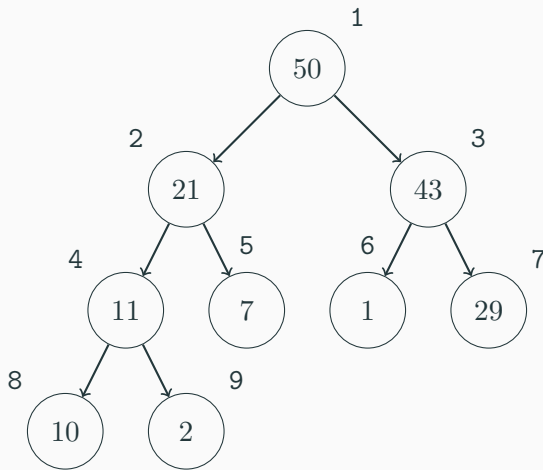
# Heaps

- As an example of a data structure in the standard library but that sometimes requires a more powerful version, let us consider heaps.
- Heaps are implemented in most standard libraries in the forms of priority queues.
- A heap is nothing but a binary tree satisfying *the heap condition*.
- The heap condition (for a min heap) says that the value of any given node is not greater than that of its children.

# Heaps

- Since arrays are linear, we want to smush this binary tree into an array for the implementation.
- We can do this by putting the root at index 1. Then the children of item at index  $i$  are simply at  $2i$  and  $2i + 1$ . The parent of any item  $i > 1$  is then  $\lfloor i/2 \rfloor$ .
- We could do this using raw arrays (then index 0 can be used to store its size), but the examples will be given in C++ using vectors.

# Heaps



ARRAY: [SIZE, 50, 21, 43, 11, 7, 1, 29, 10, 2]

# Heaps

- Items can be inserted by pushing them to the back and fixing the heap condition upwards from them.
- Items can be deleted by replacing the smallest value with a leaf and then fixing the heap condition downwards.
- Let us see how this would look in C++.



# C++ implementation (min-heap)

<MINTED>

# Heaps

- We note that peek and size run in  $\mathcal{O}(1)$  while all other operations run in  $\mathcal{O}(\log(n))$ .
- This implementation isn't any better than the standard library one in C++.
- But let us consider a harder problem where the standard heap (and this one) aren't good enough.
- This particular implementation won't be needed for problems in the course, it is merely an example.

## Hard heap problem (Trade Routes on Kattis)

- You are given a tree on  $n \leq 300\,000$  vertices.
- Each of the nodes want to trade with Rome, located at node 1. A trade route from that node puts strain on all nodes on the way from that node to Rome.
- Each node has a trade value which is how much would be gained from it trading with Rome.
- Each node has a capacity which is the maximum strain it can tolerate from trade routes.
- How much trade value can be gained at most?

# How to use heaps

- Imagine each node has a heap.
- At the start each heap just contains the trade value at that node.
- Then we move from the leaves inwards, merging together the heaps from children to parents as we go.
- This may make the heap larger than the capacity at some point, so we pop values from it until this is fixed.
- This would make the final heap at Rome contain the trade routes we want.

## How to use heaps (?)

- There is just one problem.
- The STL (standard library) implementation has a merging operation that runs in  $\mathcal{O}(n)$ , so this algorithm would be  $\mathcal{O}(n^2)$  which is far too slow ( $\sim 15$  minute runtime).
- Can we make our own heap that merges in  $\mathcal{O}(\log(n))$  or better?

## How to use heaps (?)

- There is just one problem.
- The STL (standard library) implementation has a merging operation that runs in  $\mathcal{O}(n)$ , so this algorithm would be  $\mathcal{O}(n^2)$  which is far too slow ( $\sim 15$  minute runtime).
- Can we make our own heap that merges in  $\mathcal{O}(\log(n))$  or better?
- The reason the problem is so hard is that this implementation is rather involved. I'll put it here for completeness's sake, but we will not delve much deeper here.

# Pairing heap C++

<MINTED>

<MINTED>

## How to use (fancy) heaps

- This heap can also be used wherever you'd use the STL one as well.
- This one can peek, insert and merge in  $\mathcal{O}(1)$  and pop in  $\mathcal{O}(\log(n))$ .
- It has more overhead though, so in practice it will be a fair bit slower than the STL one for non-merge operations.