
Final Review

WHAT WE LEARNED...

C++ features

Recursive thinking

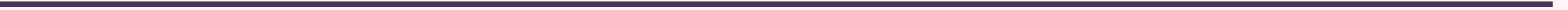
Containers

Elementary Algorithms

C++ FEATURES

C++ FEATURES

- Functions



C++ FEATURES

- Functions
 - declarations and definitions, header files

C++ FEATURES

- Functions
 - declarations and definitions, header files
 - passing by reference, especially by const reference

C++ FEATURES

- Functions
 - declarations and definitions, header files
 - passing by reference, especially by const reference
- Strings: basic operations and member functions

C++ FEATURES

- Functions
 - declarations and definitions, header files
 - passing by reference, especially by const reference
- Strings: basic operations and member functions
- Iterators

C++ FEATURES

- Functions
 - declarations and definitions, header files
 - passing by reference, especially by const reference
 - Strings: basic operations and member functions
 - Iterators
 - Struct
-

C++ FEATURES

- Functions
 - declarations and definitions, header files
 - passing by reference, especially by const reference
 - Strings: basic operations and member functions
 - Iterators
 - Struct
 - Containers and algorithms from the standard library
-

C++ FEATURES

- Functions
 - declarations and definitions, header files
 - passing by reference, especially by const reference
 - Strings: basic operations and member functions
 - Iterators
 - Struct
 - Containers and algorithms from the standard library
 - Bit operation (optional)
-

RECURSIVE THINKING

RECURSIVE THINKING

- Recipe: base case + recursive step

RECURSIVE THINKING

- Recipe: base case + recursive step
- Analysis: Induction (base case + inductive step)

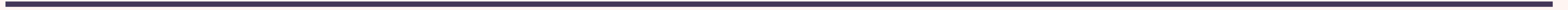
RECURSIVE THINKING

- Recipe: base case + recursive step
- Analysis: Induction (base case + inductive step)
- Recursive solution of problems



RECURSIVE THINKING

- Recipe: base case + recursive step
- Analysis: Induction (base case + inductive step)
- Recursive solution of problems
 - Exhaustive search



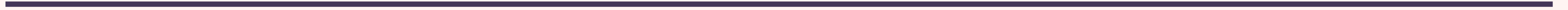
RECURSIVE THINKING

- Recipe: base case + recursive step
- Analysis: Induction (base case + inductive step)
- Recursive solution of problems
 - Exhaustive search
 - Permutation, combination



RECURSIVE THINKING

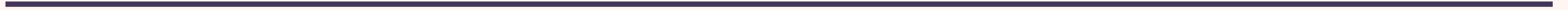
- Recipe: base case + recursive step
- Analysis: Induction (base case + inductive step)
- Recursive solution of problems
 - Exhaustive search
 - Permutation, combination
 - Backtracking



SEQUENTIAL CONTAINERS

SEQUENTIAL CONTAINERS

- Vector



SEQUENTIAL CONTAINERS

- Vector
- Stack

- Queue
- Deque

- List
- LinkedList

- ArrayList
- Vector

- LinkedList
- Deque

- Queue
- Deque

- Queue
 - Deque
-

SEQUENTIAL CONTAINERS

- Vector
- Stack
 - First in first out



SEQUENTIAL CONTAINERS

- Vector
- Stack
 - First in first out
 - Parenthesis pairing



SEQUENTIAL CONTAINERS

- Vector
- Stack
 - First in first out
 - Parenthesis pairing
 - push, pop, top



SEQUENTIAL CONTAINERS

- Vector
- Stack
 - First in first out
 - Parenthesis pairing
 - push, pop, top
- Queue

SEQUENTIAL CONTAINERS

- Vector
- Stack
 - First in first out
 - Parenthesis pairing
 - push, pop, top
- Queue
 - First in last out

SEQUENTIAL CONTAINERS

- Vector
- Stack
 - First in first out
 - Parenthesis pairing
 - push, pop, top
- Queue
 - First in last out
 - Used in BFS

SEQUENTIAL CONTAINERS

- Vector
- Stack
 - First in first out
 - Parenthesis pairing
 - push, pop, top
- Queue
 - First in last out
 - Used in BFS
 - push, pop, front

SEQUENTIAL CONTAINERS

- Vector
- Stack
 - First in first out
 - Parenthesis pairing
 - push, pop, top
- Queue
 - First in last out
 - Used in BFS
 - push, pop, front
- Deque

SEQUENTIAL CONTAINERS

- Vector
 - Stack
 - First in first out
 - Parenthesis pairing
 - push, pop, top
 - Queue
 - First in last out
 - Used in BFS
 - push, pop, front
 - Deque
 - push_back, push_front, pop_back, pop_front, front, back
-

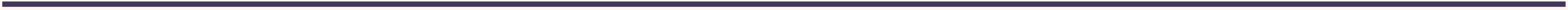
ASSOCIATIVE CONTAINERS

ASSOCIATIVE CONTAINERS

- Set

ASSOCIATIVE CONTAINERS

- Set
 - Collection of sortable objects



ASSOCIATIVE CONTAINERS

- Set
 - Collection of sortable objects
- Map



ASSOCIATIVE CONTAINERS

- Set
 - Collection of sortable objects
- Map
 - Collection of <keyword, value> pairs

ASSOCIATIVE CONTAINERS

- Set
 - Collection of sortable objects
- Map
 - Collection of <keyword, value> pairs
- These are "associative containers"



ASSOCIATIVE CONTAINERS

- Set
 - Collection of sortable objects
- Map
 - Collection of <keyword, value> pairs
- These are "associative containers"
- Basic operations (except traversal) are all in time $O(\log n)$



ASSOCIATIVE CONTAINERS

- Set
 - Collection of sortable objects
- Map
 - Collection of <keyword, value> pairs
- These are "associative containers"
- Basic operations (except traversal) are all in time $O(\log n)$
 - lookup



ASSOCIATIVE CONTAINERS

- Set
 - Collection of sortable objects
 - Map
 - Collection of <keyword, value> pairs
 - These are "associative containers"
 - Basic operations (except traversal) are all in time $O(\log n)$
 - lookup
 - insertion, deletion
-

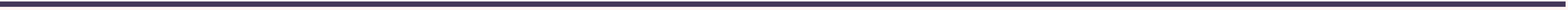
ASSOCIATIVE CONTAINERS

- Set
 - Collection of sortable objects
 - Map
 - Collection of <keyword, value> pairs
 - These are "associative containers"
 - Basic operations (except traversal) are all in time $O(\log n)$
 - lookup
 - insertion, deletion
 - next larger, next smaller
-

ITERATORS

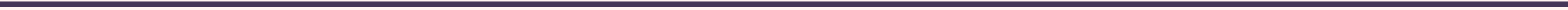
ITERATORS

- Iterators allow us to iterate over elements in STL containers



ITERATORS

- Iterators allow us to iterate over elements in STL containers
 - They are essentially pointers in STL



ITERATORS

- Iterators allow us to iterate over elements in STL containers
 - They are essentially pointers in STL
- STL functions that return iterators: `begin()`, `end()`, `find()`, `lower_bound()`, `upper_bound()`



ITERATORS

- Iterators allow us to iterate over elements in STL containers
 - They are essentially pointers in STL
 - STL functions that return iterators: `begin()`, `end()`, `find()`, `lower_bound()`, `upper_bound()`
 - Operations: `==`, `!=`, `++`, `--` (for all containers), `i + 4`, `i - j` (only for vector and deque)
-

ITERATORS

- Iterators allow us to iterate over elements in STL containers
 - They are essentially pointers in STL
 - STL functions that return iterators: `begin()`, `end()`, `find()`, `lower_bound()`, `upper_bound()`
 - Operations: `==`, `!=`, `++`, `--` (for all containers), `i + 4`, `i - j` (only for vector and deque)
 - Iterator range: `[Iterator1, Iterator2)`
-

ITERATORS

- Iterators allow us to iterate over elements in STL containers
 - They are essentially pointers in STL
 - STL functions that return iterators: `begin()`, `end()`, `find()`, `lower_bound()`, `upper_bound()`
 - Operations: `==`, `!=`, `++`, `--` (for all containers), `i + 4`, `i - j` (only for vector and deque)
 - Iterator range: `[Iterator1, Iterator2)`
 - Many STL algorithms use iterator range to specify its domain: `sort()`, `lower_bound()`, `upper_bound()`, `nth_element()`...
-

ALGORITHM ANALYSIS

ALGORITHM ANALYSIS

ALGORITHM ANALYSIS

- Analysis of running time of an algorithm

ALGORITHM ANALYSIS

- Analysis of running time of an algorithm
 - Basic idea: worst case running time as a function of the input length

ALGORITHM ANALYSIS

- Analysis of running time of an algorithm
 - Basic idea: worst case running time as a function of the input length
 - $O(\cdot)$ notation

ALGORITHM ANALYSIS

- Analysis of running time of an algorithm
 - Basic idea: worst case running time as a function of the input length
 - $O(\cdot)$ notation
 - Derivation of the simplest running time: $O(n)$, $O(n^2)$, $O(n \log n)$...



ALGORITHM ANALYSIS

- Analysis of running time of an algorithm
 - Basic idea: worst case running time as a function of the input length
 - $O(\cdot)$ notation
 - Derivation of the simplest running time: $O(n)$, $O(n^2)$, $O(n \log n)$...
- Sorting



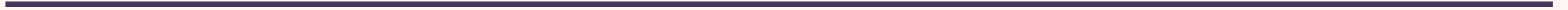
ALGORITHM ANALYSIS

- Analysis of running time of an algorithm
 - Basic idea: worst case running time as a function of the input length
 - $O(\cdot)$ notation
 - Derivation of the simplest running time: $O(n)$, $O(n^2)$, $O(n \log n)$...
- Sorting
 - Insertion sort, runtime $O(n^2)$



ALGORITHM ANALYSIS

- Analysis of running time of an algorithm
 - Basic idea: worst case running time as a function of the input length
 - $O(\cdot)$ notation
 - Derivation of the simplest running time: $O(n)$, $O(n^2)$, $O(n \log n)$...
- Sorting
 - Insertion sort, runtime $O(n^2)$
 - Merge sort, runtime $O(n \log n)$



ALGORITHM ANALYSIS

- Analysis of running time of an algorithm
 - Basic idea: worst case running time as a function of the input length
 - $O(\cdot)$ notation
 - Derivation of the simplest running time: $O(n)$, $O(n^2)$, $O(n \log n)$...
 - Sorting
 - Insertion sort, runtime $O(n^2)$
 - Merge sort, runtime $O(n \log n)$
 - Counting sort, runtime $O(n)$ (but only for entries from a small universe)
-

SORTING

SORTING

SORTING

- Sorting



SORTING

- Sorting
 - Bubble sort and insertion sort, runtime $O(n^2)$



SORTING

- Sorting
 - Bubble sort and insertion sort, runtime $O(n^2)$
 - Merge sort, runtime $O(n \log n)$



SORTING

- Sorting
 - Bubble sort and insertion sort, runtime $O(n^2)$
 - Merge sort, runtime $O(n \log n)$
 - A lower bound of $\Omega(n \log n)$ running time for comparison based sorting



SORTING

- Sorting
 - Bubble sort and insertion sort, runtime $O(n^2)$
 - Merge sort, runtime $O(n \log n)$
 - A lower bound of $\Omega(n \log n)$ running time for comparison based sorting
 - Counting sort, runtime $O(n)$ (only when entries are from a small universe)



SORTING

- Sorting
 - Bubble sort and insertion sort, runtime $O(n^2)$
 - Merge sort, runtime $O(n \log n)$
 - A lower bound of $\Omega(n \log n)$ running time for comparison based sorting
 - Counting sort, runtime $O(n)$ (only when entries are from a small universe)
 - To use `sort()` from STL, sometimes a self-defined comparator function may be needed
-

BINARY SEARCH

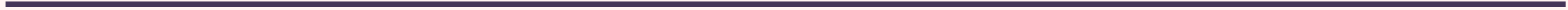
BINARY SEARCH

- Basic binary search: on *sorted* containers



BINARY SEARCH

- Basic binary search: on *sorted* containers
 - STL functions can be used for this: `binary_search()`, `lower_bound()`, `upper_bound()`



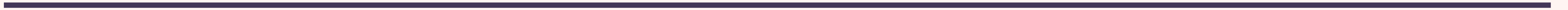
BINARY SEARCH

- Basic binary search: on *sorted* containers
 - STL functions can be used for this: `binary_search()`, `lower_bound()`, `upper_bound()`
 - On sequential containers, this can be used to count the number of elements in $[a, b)$:
`upper_bound(b, iter1, iter2) - lower_bound(a, iter1, iter2)`



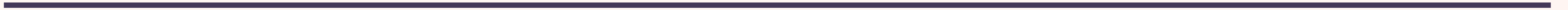
BINARY SEARCH

- Basic binary search: on *sorted* containers
 - STL functions can be used for this: `binary_search()`, `lower_bound()`, `upper_bound()`
 - On sequential containers, this can be used to count the number of elements in $[a, b)$:
`upper_bound(b, iter1, iter2) - lower_bound(a, iter1, iter2)`
- For many computational problems, binary search finds the boundary of feasible regions



BINARY SEARCH

- Basic binary search: on *sorted* containers
 - STL functions can be used for this: `binary_search()`, `lower_bound()`, `upper_bound()`
 - On sequential containers, this can be used to count the number of elements in $[a, b)$:
`upper_bound(b, iter1, iter2) - lower_bound(a, iter1, iter2)`
- For many computational problems, binary search finds the boundary of feasible regions
 - I.e., binary search is often used to reduce an optimization problem to a decision problem



BINARY SEARCH

- Basic binary search: on *sorted* containers
 - STL functions can be used for this: `binary_search()`, `lower_bound()`, `upper_bound()`
 - On sequential containers, this can be used to count the number of elements in $[a, b)$:
`upper_bound(b, iter1, iter2) - lower_bound(a, iter1, iter2)`
- For many computational problems, binary search finds the boundary of feasible regions
 - I.e., binary search is often used to reduce an optimization problem to a decision problem
 - E.g., solving for roots of a polynomial

ELEMENTARY ALGORITHMS

ELEMENTARY ALGORITHMS

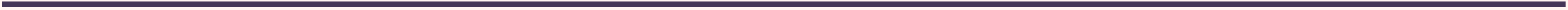
- Two pointers

- Two pointers

- Two pointers
-

ELEMENTARY ALGORITHMS

- Two pointers
 - Merging sorted vectors



ELEMENTARY ALGORITHMS

- Two pointers
 - Merging sorted vectors
 - Shortest interval whose sum exceeds a target

Two pointers

Merging sorted vectors

Shortest interval whose sum exceeds a target

ELEMENTARY ALGORITHMS

- Two pointers
 - Merging sorted vectors
 - Shortest interval whose sum exceeds a target
 - Often used to accelerate naive $O(n^2)$ searches to $O(n)$ time



ELEMENTARY ALGORITHMS

- Two pointers
 - Merging sorted vectors
 - Shortest interval whose sum exceeds a target
 - Often used to accelerate naive $O(n^2)$ searches to $O(n)$ time
- Prefix sum



ELEMENTARY ALGORITHMS

- Two pointers
 - Merging sorted vectors
 - Shortest interval whose sum exceeds a target
 - Often used to accelerate naive $O(n^2)$ searches to $O(n)$ time
- Prefix sum
 - Linear time pre-processing, then used to compute any interval sum in $O(1)$ time



ELEMENTARY ALGORITHMS

- Two pointers
 - Merging sorted vectors
 - Shortest interval whose sum exceeds a target
 - Often used to accelerate naive $O(n^2)$ searches to $O(n)$ time
- Prefix sum
 - Linear time pre-processing, then used to compute any interval sum in $O(1)$ time
- "Difference method"



ELEMENTARY ALGORITHMS

- Two pointers
 - Merging sorted vectors
 - Shortest interval whose sum exceeds a target
 - Often used to accelerate naive $O(n^2)$ searches to $O(n)$ time
- Prefix sum
 - Linear time pre-processing, then used to compute any interval sum in $O(1)$ time
- "Difference method"
 - Compute cumulative results of interval updates



ELEMENTARY ALGORITHMS

- Two pointers
 - Merging sorted vectors
 - Shortest interval whose sum exceeds a target
 - Often used to accelerate naive $O(n^2)$ searches to $O(n)$ time
 - Prefix sum
 - Linear time pre-processing, then used to compute any interval sum in $O(1)$ time
 - "Difference method"
 - Compute cumulative results of interval updates
 - For m interval updates on a vector of length n , runtime is $O(m + n)$
-

ELEMENTARY ALGORITHMS

ELEMENTARY ALGORITHMS

- Monotone stack

- Monotone queue

- Sliding window
-

ELEMENTARY ALGORITHMS

- Monotone stack
 - Compute for each element in a vector the next element larger than it, in time $O(n)$



ELEMENTARY ALGORITHMS

- Monotone stack
 - Compute for each element in a vector the next element larger than it, in time $O(n)$
- Monotone deque



ELEMENTARY ALGORITHMS

- Monotone stack
 - Compute for each element in a vector the next element larger than it, in time $O(n)$
- Monotone deque
 - Compute maximum elements in a sliding window, in time $O(n)$



ELEMENTARY ALGORITHMS

- Monotone stack
 - Compute for each element in a vector the next element larger than it, in time $O(n)$
- Monotone deque
 - Compute maximum elements in a sliding window, in time $O(n)$
- Dynamic programming



ELEMENTARY ALGORITHMS

- Monotone stack
 - Compute for each element in a vector the next element larger than it, in time $O(n)$
- Monotone deque
 - Compute maximum elements in a sliding window, in time $O(n)$
- Dynamic programming
 - Reuse solutions to subproblems by storing them. Essence: recursion with memory



ELEMENTARY ALGORITHMS

- Monotone stack
 - Compute for each element in a vector the next element larger than it, in time $O(n)$
- Monotone deque
 - Compute maximum elements in a sliding window, in time $O(n)$
- Dynamic programming
 - Reuse solutions to subproblems by storing them. Essence: recursion with memory
 - Select a subset from towers on a line with constraints



ELEMENTARY ALGORITHMS

- Monotone stack
 - Compute for each element in a vector the next element larger than it, in time $O(n)$
 - Monotone deque
 - Compute maximum elements in a sliding window, in time $O(n)$
 - Dynamic programming
 - Reuse solutions to subproblems by storing them. Essence: recursion with memory
 - Select a subset from towers on a line with constraints
 - Find a longest increasing subsequence in a given vector (in time $O(n)$)
-

ELEMENTARY ALGORITHMS

- Monotone stack
 - Compute for each element in a vector the next element larger than it, in time $O(n)$
 - Monotone deque
 - Compute maximum elements in a sliding window, in time $O(n)$
 - Dynamic programming
 - Reuse solutions to subproblems by storing them. Essence: recursion with memory
 - Select a subset from towers on a line with constraints
 - Find a longest increasing subsequence in a given vector (in time $O(n)$)
 - Backpacking (when each item has one copy / multiple copies / unlimited copies)
-

ELEMENTARY ALGORITHMS

- Monotone stack
 - Compute for each element in a vector the next element larger than it, in time $O(n)$
 - Monotone deque
 - Compute maximum elements in a sliding window, in time $O(n)$
 - Dynamic programming
 - Reuse solutions to subproblems by storing them. Essence: recursion with memory
 - Select a subset from towers on a line with constraints
 - Find a longest increasing subsequence in a given vector (in time $O(n)$)
 - Backpacking (when each item has one copy / multiple copies / unlimited copies)
 - Sequential decision making with states
-