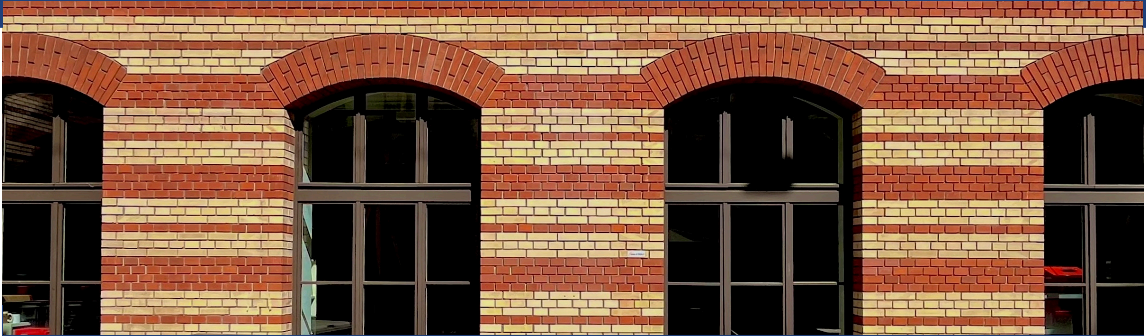


Datastructures and Algorithms

Generic Programming, Higher-Order Functions, Convex Hull

Adel Gavranović — ETH Zürich — 2025

Overview

Learning Objectives
Generic Programming: Higher Order Functions
    Function Signature Notation
Convex Hull
Past Exam Questions

n.ethz.ch/~agavranovic

Material

Webpage

Mail

1. Follow-up

←

Follow-up from last session

Follow-up from last session

- I’m back! (if you liked Tristan’s style, you’re free to move to his class)

Follow-up from last session

- I’m back! (if you liked Tristan’s style, you’re free to move to his class)
- Need more assistance?

Follow-up from last session

- I’m back! (if you liked Tristan’s style, you’re free to move to his class)
- Need more assistance?
  - Visit the Study Center (esp. for the bonus exercise!)
  - Read Course Script
  - Check out the Moodle Forum
  - Write me an e-mail
    Always include your code and what you’ve tried thus far

2. Feedback regarding **code** expert

General things regarding **code** expert

General things regarding **code** expert

- Amazing submission in LaTeX!

General things regarding code expert

- Amazing submission in $\LaTeX$!
- If you spot any issues with my corrections (e.g. too strict, mismatched rating and feedback), call me out on it ASAP via e-mail

General things regarding **code** expert

- Amazing submission in LaTeX!
- If you spot any issues with my corrections (e.g. too strict, mismatched rating and feedback), call me out on it ASAP via e-mail
- If you're ever very close to unlocking an exercise and only have uncorrected points from theory exercises left, then also write me an e-mail

General things regarding **code** expert

- Amazing submission in LaTeX!
- If you spot any issues with my corrections (e.g. too strict, mismatched rating and feedback), call me out on it ASAP via e-mail
- If you're ever very close to unlocking an exercise and only have uncorrected points from theory exercises left, then also write me an e-mail
- How difficult are the (weekly) exercises to you?

↳ descriptions suck

↳ loaded code has issues

↳ sometimes unclear re: which libs are allowed

←

Any questions regarding **code** expert on your part?

3. Learning Objectives

Objectives

Objectives

- ☐ Understand what 'Callables' are
- ☐ Understand what Higher-Order Functions are and what they're used for
- ☐ Understand why and how the Jarvis March Algorithm works
- ☐ Be able to implement the Jarvis March Algorithm

4. Summary

←

Getting on the same page

Getting on the same page

- What did you see in the lectures?

5. Generic Programming: Higher Order Functions

Motivation

- Overarching goal: make code generic, thus reusable

Motivation

- Overarching goal: make code generic, thus reusable
- Templates so far: make code parametric in the data it operates on, e.g.
  - `Pair<T>` for all types `T`
  - `print<C>` for all iterable containers `C`

Motivation

- Overarching goal: make code generic, thus reusable
- Templates so far: make code parametric in the data it operates on, e.g.
  - `Pair<T>` for all types `T`
  - `print<C>` for all iterable containers `C`
- Now: make code parametric in the algorithms it uses, e.g.
  - `filter(container, predicate)`
  - `apply(signal, transformation/filter)`
  - `leader_election(participants, protocol)`
  - `navigation_system(map, shortest_path_algorithm)`
  - `Button("Save").onClick(handle_click_event)`

Callables and Higher-Order Functions

```cpp
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;

  for (const auto& e : src_data)
    if (pred(e)) {
      data.push_back(e);
    }

  return data;
}
```

←

Callables and Higher-Order Functions

```
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;

  for (const auto& e : src_data)
    if (pred(e)) {
      data.push_back(e);
    }

  return data;
}
```

- `pred` must be callable (applicable, invocable), i.e., something function-like

f(), f(), λ

Callables and Higher-Order Functions

```
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;

  for (const auto& e : src_data)
    if (pred(e)) {
      data.push_back(e);
    }

  return data;
}
```

"range-based for-loop"

cont.

- `pred` must be callable (applicable, invocable), i.e., something function-like
- In C++:
  - free or member function
  - lambda function
  - functor (object with `operator()`)
  - `std::function` object
  - function pointers [not discussed]

auto it = src_data.begin();

it++

Callables and Higher-Order Functions

```cpp
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;

  for (const auto& e : src_data)
    if (pred(e)) {
      data.push_back(e);
    }

  return data;
}
```

- `pred` must be callable (applicable, invocable), i.e., something function-like
- In C++:
  - free or member function
  - lambda function
  - functor (object with `operator()`)
  - `std::function` object
  - function pointers [not discussed]

Functions taking or returning functions are called **higher-order functions**.

C++Functors

```
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;
  for (const auto& e : src_data)
    if (pred(e)) data.push_back(e);
  return data;
}
```

```
// stateful predicate as functor
template <typename T>
struct AtLeast {
  T min;
  AtLeast(T m): min(m) {};
  bool operator()(T i) const {
    return min <= i;
  }
};
```

t.operator(v);

C++Functors

```cpp
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;
  for (const auto& e : src_data)
    if (pred(e)) data.push_back(e);
  return data;
}
```

```cpp
// stateful predicate as functor
template <typename T>
struct AtLeast {
  T min;
  AtLeast(T m): min(m) {};
  bool operator()(T i) const {
    return min <= i;
  }
};
```

- A functor
  - is an object that implements `operator()`
  - combines state (since an object) with callability (since `operator()`)

"memory"

C++Functors

```cpp
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;
  for (const auto& e : src_data)
    if (pred(e)) data.push_back(e);
  return data;
}
```

```cpp
// stateful predicate as functor
template <typename T>
struct AtLeast {
  T min;
  AtLeast(T m): min(m) {};
  bool operator()(T i) const {
    return min <= i;
  }
};
```

- A functor
  - is an object that implements `operator()`
  - combines state (since an object) with callability (since `operator()`)
- Objects of type `AtLeast<T>` are callable with one `T` argument.

C++Functors

```cpp
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;
  for (const auto& e : src_data)
    if (pred(e)) data.push_back(e);
  return data;
}
```

```cpp
// stateful predicate as functor
template <typename T>
struct AtLeast {
  T min;
  AtLeast(T m): min(m) {};
  bool operator()(T i) const {
    return min <= i;
  }
};
```

- A functor bool
  - is an object that implements `operator()`
  - combines state (since an object) with callability (since `operator()`)
- Objects of type `AtLeast<T>` are callable with one `T` argument.

```cpp
std::vector<int> data = {-1,0,1,2,-2,4,-3};
selection1 = filter(data, AtLeast(-1));
// = {-1,0,1,2,4}
selection2 = filter(data, AtLeast(4));
// = {4,
```

Lambda Expressions Translate to Functors

```
std::vector<int> data = {-1,0,1,2,-2,4,5,-3};

auto selection1 = filter(data, [](int e) { return -2 <= e; });
auto selection2 = filter(data, [](int e) { return e != 0; });
```

```
struct lambda1 {
  bool operator()(int e) const {
    return -2 <= e;
  }
};

struct lambda2 {
  bool operator()(int e) const {
    return e != 0;
  }
};
```

Lambda Expressions Translate to Functors

```cpp
std::vector<int> data = {-1,0,1,2,-2,4,5,-3};

auto selection1 = filter(data, [](int e) { return -2 <= e; });
auto selection2 = filter(data, [](int e) { return e != 0; });
```

```cpp
struct lambda1 {
  bool operator()(int e) const {
    return -2 <= e;
  }
};

struct lambda2 {
  bool operator()(int e) const {
    return e != 0;
  }
};
```

- C++compiler generates functors from lambda expressions

Lambda Expressions Translate to Functors

```
std::vector<int> data = {-1,0,1,2,-2,4,5,-3};

auto selection1 = filter(data, [](int e) { return -2 <= e; });
auto selection2 = filter(data, [](int e) { return e != 0; });
```

```
struct lambda1 {
  bool operator()(int e) const {
    return -2 <= e;
  }
};

struct lambda2 {
  bool operator()(int e) const {
    return e != 0;
  }
};
```

- C++compiler generates functors from lambda expressions
- Lambdas are not essential, but “merely” convenient

Lambda Expression Syntax

Most general shape:

$$\text{``}\ \underbrace{[e_1, \ldots, e_n]}_{\text{captures}}\ \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}}\ \underbrace{\texttt{-> R}}_{\text{return type}}\ \underbrace{\texttt{\{ stmt \}}}_{\text{body}}\ \text{''}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}}\ \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}}\ \underbrace{\texttt{->}\ \texttt{R}}_{\text{return type}}\ \underbrace{\texttt{\{ stmt \}}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \; \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \; \underbrace{\texttt{->}\ \texttt{R}}_{\text{return type}} \; \underbrace{\{\ \texttt{stmt}\ \}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)
- `[&x]` `x` is accessible by reference

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \ \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \ \underbrace{\texttt{->}\ \texttt{R}}_{\text{return type}} \ \underbrace{\texttt{\{ stmt \}}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)
- `[&x]` `x` is accessible by reference
- `[x, &y]` `x` is copied, `y` is referenced

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \; \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \; \underbrace{\texttt{->}\ \texttt{R}}_{\text{return type}} \; \underbrace{\texttt{\{ stmt \}}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)
- `[&x]` `x` is accessible by reference
- `[x, &y]` `x` is copied, `y` is referenced
- `[&]` all necessary variables are automatically referenced

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \; \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \; \underbrace{\texttt{->\ R}}_{\text{return type}} \; \underbrace{\texttt{\{ stmt \}}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)
- `[&x]` `x` is accessible by reference
- `[x, &y]` `x` is copied, `y` is referenced
- `[&]` all necessary variables are automatically referenced
- `[=]` all necessary variables are automatically copied

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \; \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \; \underbrace{\texttt{->}\ \texttt{R}}_{\text{return type}} \; \underbrace{\texttt{\{ stmt \}}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)
- `[&x]` `x` is accessible by reference
- `[x, &y]` `x` is copied, `y` is referenced
- `[&]` all necessary variables are automatically referenced
- `[=]` all necessary variables are automatically copied
- `[&, x]` all necessary variables are referenced, except `x`, which is copied

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \ \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \ \underbrace{\texttt{->}\ \texttt{R}}_{\text{return type}} \ \underbrace{\texttt{\{ stmt \}}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)
- `[&x]` `x` is accessible by reference
- `[x, &y]` `x` is copied, `y` is referenced
- `[&]` all necessary variables are automatically referenced
- `[=]` all necessary variables are automatically copied
- `[&, x]` all necessary variables are referenced, except `x`, which is copied
- `[=, &x]` all necessary variables are copied, except `x`, which is referenced

Functors

□ reading scope of capture

1. Write down the functor that corresponds to the lambda
2. Use the functor in the `filter()` expression

```
int count = 0;
int min = 3;
std::vector<int> data = {4,-2,0};
data = filter(data, [&, min](int e) {
  ++count; return min <= e;
});
```

class lambda1 {

fill me

};

int min

int count

Functors

1. Write down the functor that corresponds to the lambda
2. Use the functor in the `filter()` expression

```
int count = 0;
int min = 3;
std::vector<int> data = {4,-2,0};
data = filter(data, [&, min](int e) {
  ++count; return min <= e;
});
```

Solution

why const?

```
class lambda1 {
  int& count;
  const int min;
public:
  lambda1(int& c, int m):
    count(c), min(m) {}
  bool operator()(int e) const {
    ++count;
    return min <= e;
  }
};
```

```
int count = 0;
int min = 3;
std::vector<int> data = {4,-2,0};

data = filter(data, lambda1(count, min));
```

// count == 3;

Functors

- Observe that the lambda now uses the `auto` type placeholder for its argument

```
data = filter(data, [](auto e) { return 0 <= e; });
```

Functors

- Observe that the lambda now uses the `auto` type placeholder for its argument

```
data = filter(data, [](auto e) { return 0 <= e; });
```

- Question: How is this reflected by the generated functor?

Functors

- Observe that the lambda now uses the `auto` type placeholder for its argument

```
data = filter(data, [](auto e) { return 0 <= e; });
```

- Question: How is this reflected by the generated functor?
- Solution:

```
class lambda2 {
public:
  lambda2() {}

  template <typename T>
  bool operator()(T e) const {
    return 0 <= e;
  }
};
```

X

5. Generic Programming: Higher Order Functions

5.1. Function Signature Notation

not exam relevant

←

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types
- Examples:

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types
- Examples:
  - $f_1 : A \rightarrow \text{int}$ function from any type to integer

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types
- Examples:
  - $f_1 : A \to \text{int}$ function from any type to integer
  - $f_2 : A \times A \times A \to \text{bool}$ function from three A's to boolean

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types
- Examples:
  - $f_1 : A \rightarrow \text{int}$ function from any type to integer
  - $f_2 : A \times A \times A \rightarrow \text{bool}$ function from three A's to boolean
  - $f_3 : A \times (A \rightarrow B) \rightarrow B$ "higher-order function" (with two arguments)

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types
- Examples:
  - $f_1 : A \to \text{int}$ function from any type to integer
  - $f_2 : A \times A \times A \to \text{bool}$ function from three A's to boolean
  - $f_3 : A \times (A \to B) \to B$ "higher-order function" (with two arguments)
  - $f_4 : \texttt{vec<A>} \times (A \to B) \to \texttt{vec<B>}$ higher-order function involving vectors

←

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types
- Examples:
  - $f_1 : A \to \text{int}$ function from any type to integer
  - $f_2 : A \times A \times A \to \text{bool}$ function from three A's to boolean
  - $f_3 : A \times (A \to B) \to B$ "higher-order function" (with two arguments)
  - $f_4 : \texttt{vec<A>} \times (A \to B) \to \texttt{vec<B>}$ higher-order function involving vectors
  - $f_5 : (A \times A \to B) \times A \to ((A \to B) \to \text{bool})$ taking and returning a function

Function Signature Notation: Example 1

- Task: Write down a function with signature $f_2 : A \times A \to \text{bool}$

Function Signature Notation: Example 1

- Task: Write down a function with signature $f_2 : A \times A \to \text{bool}$
- Solution:

```
template <typename A>
bool eq(A a1, A a2) {
  return a1 == a2;
}
```

←

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- Solution 1:

```
template <typename A, typename F>
auto apply1(A a, F a_to_b) {
  return a_to_b(a);
}

int i1 = apply1('a', [](char c) { return c - 65; });
```

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- Solution 1:

```
template <typename A, typename F>
auto apply1(A a, F a_to_b) {
  return a_to_b(a);
}

int i1 = apply1('a', [](char c) { return c - 65; });
```

- Observations
  - type parameter `B` is only implicitly given, as `F`'s return type
  - template type parameters inferred at call-site

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- Solution 2:

```cpp
template <typename A, typename B>
B apply2(A a, std::function<B(A)> a_to_b) {
  return a_to_b(a);
}

int i2 = apply2('a', std::function([](char c) { return c - 65; }));
```

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- Solution 2:

```
template <typename A, typename B>
B apply2(A a, std::function<B(A)> a_to_b) {
  return a_to_b(a);
}

int i2 = apply2('a', std::function([](char c) { return c - 65; }));
```

- Observations
  - type parameter `B` is explicit
  - but we need to wrap the lambda in a **`std::function`**
  - template type parameters inferred at call-site

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- ~~Solution~~ Attempt 3

```
template <typename A, typename F, typename B>
B apply3(A a, F a_to_b) {
  return a_to_b(a);
}

int i3 = apply3<char, ???, int>('a', [](char c) { return c - 65; });
```

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- ~~Solution~~ Attempt 3

```
template <typename A, typename F, typename B>
B apply3(A a, F a_to_b) {
  return a_to_b(a);
}

int i3 = apply3<char, ???, int>('a', [](char c) { return c - 65; });
```

- Observations
  - type parameter `B` is explicit
  - but not directly connected to return type of `F`

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- ~~Solution~~ Attempt 3

```
template <typename A, typename F, typename B>
B apply3(A a, F a_to_b) {
  return a_to_b(a);
}

int i3 = apply3<char, ???, int>('a', [](char c) { return c - 65; });
```

- Observations
  - type parameter `B` is explicit
  - but not directly connected to return type of `F`
- Problem: At call-site, `B` can't be inferred. We can explicitly instantiate `B` - but now we'd have to do that for `F` as well, which we can't.

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind(planet, [](auto s1, auto s2) { return s1 + s2; });
```

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind(planet, [](auto s1, auto s2) { return s1 + s2; });
```

- Question: how to use `f`?

←

Function Signature Notation: Example 3

- Task: Write down a function with signature $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind(planet, [](auto s1, auto s2) { return s1 + s2; });
```

- Question: how to use `f`?
- Answer:

```
std::cout << f(" is the fourth planet from the sun.");
```

←

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind(planet, [](auto s1, auto s2) { return s1 + s2; });
```

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind(planet, [](auto s1, auto s2) { return s1 + s2; });
```

Function Signature Notation: Example 3

- Task: Write down a function with signature $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind(planet, [](auto s1, auto s2) { return s1 + s2; });
```

- Question: What would happen if the capture were `[&]` instead of `[=]`?

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind(planet, [](auto s1, auto s2) { return s1 + s2; });
```

- Question: What would happen if the capture were `[&]` instead of `[=]`?

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind(planet, [](auto s1, auto s2) { return s1 + s2; });
```

- Question: What would happen if the capture were `[&]` instead of `[=]`?
  Answer: The returned lambda would capture argument `a1` by reference, but `a1` is removed from memory when the call to `bind()` terminates. Calling `f` would thus result in undefined behaviour.

A Prominent Higher Order Function

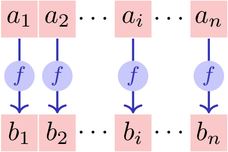
- Consider the function $m : \texttt{vec<A>} \times (A \to B) \to \texttt{vec<B>}$

A Prominent Higher Order Function

- Consider the function $m : \texttt{vec<A>} \times (A \to B) \to \texttt{vec<B>}$
- Given the signature above, what could function m do?

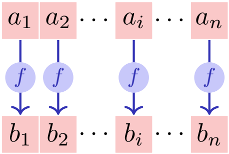
A Prominent Higher Order Function

- Consider the function $m : \texttt{vec<A>} \times (A \to B) \to \texttt{vec<B>}$
- Given the signature above, what could function m do?
- Visual hint:



A Prominent Higher Order Function

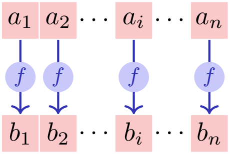
- Consider the function $m : \texttt{vec<A>} \times (A \to B) \to \texttt{vec<B>}$
- Given the signature above, what could function m do?
- Visual hint:



- Task: Implement the function in C++

A Prominent Higher Order Function

- Consider the function $m : \texttt{vec<A>} \times (A \to B) \to \texttt{vec<B>}$
- Given the signature above, what could function m do?
- Visual hint:



- Task: Implement the function in C++
- Solution:

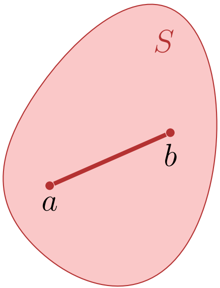
```cpp
template <typename A, typename B>
std::vector<B> map(std::vector<A> as, std::function<B(A)> f) {
  std::vector<B> result;
  for (const auto& a : as)
    result.push_back(f(a));
  return result;
}
```

6. Convex Hull

Convex Hull

Subset S of a real vector space is called **convex**, if for all $a, b \in S$ and all $\lambda \in [0, 1]$:

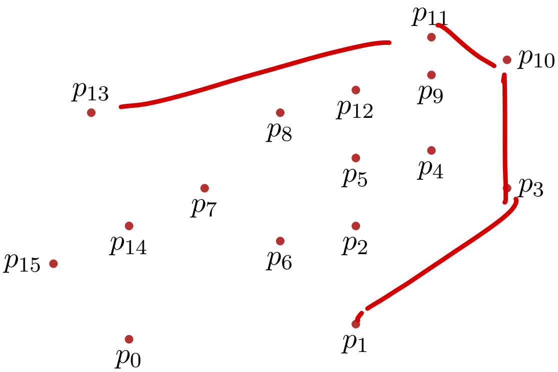
$$\lambda a + (1 - \lambda) b \in S$$



not convex

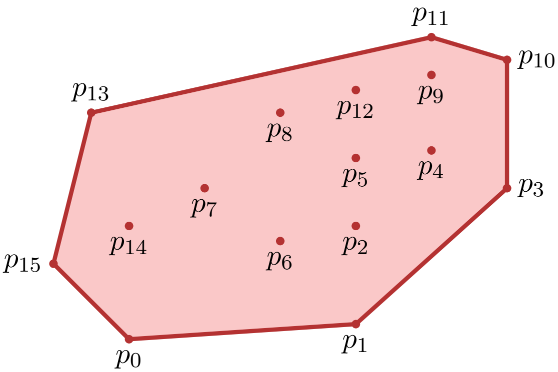
Convex Hull

Convex Hull $H(Q)$ of a set Q of points: smallest convex polygon P such that each point of Q is on P or in the interior of P.



Convex Hull

Convex Hull $H(Q)$ of a set Q of points: smallest convex polygon P such that each point of Q is on P or in the interior of P.

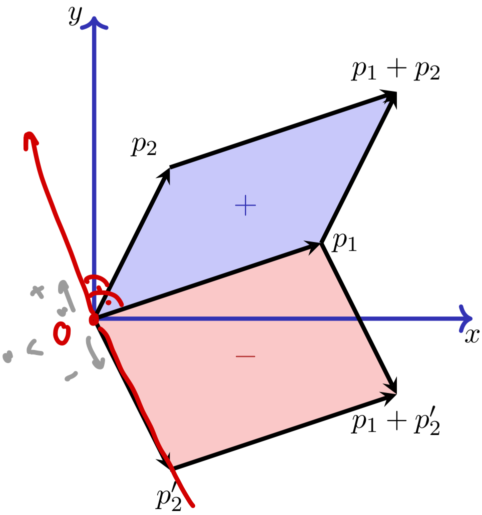


Properties of line segments

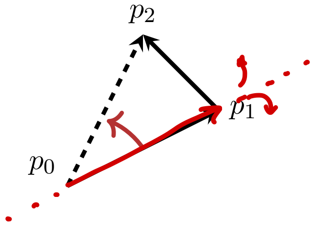
Cross-Product of two vectors $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ in the plane

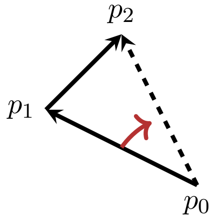
$$\| p_1 \times p_2 \| = \det \begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix} = x_1 y_2 - x_2 y_1$$

Signed area of the parallelogram



Turning direction





to the left:
$\|(p_1 - p_0) \times (p_2 - p_0)\| > 0$

to the right:
$\|(p_1 - p_0) \times (p_2 - p_0)\| < 0$

Jarvis March / Gift Wrapping algorithm

1. Start with an extremal point (e.g. lowest point) $p = p_0$
2. Search point q, such that $\overline{pq}$ is a line to the right of all other points (or other points are on this line closer to p.
3. Continue with $p \leftarrow q$ at (2) until $p = p_0$.

Illustration Jarvis

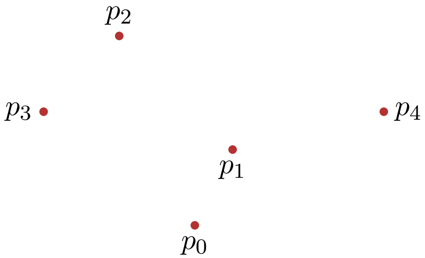
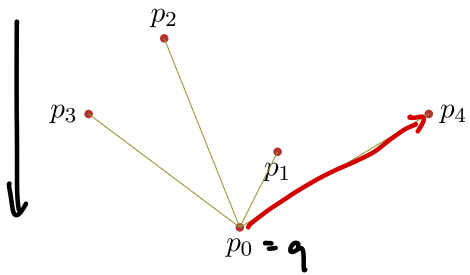
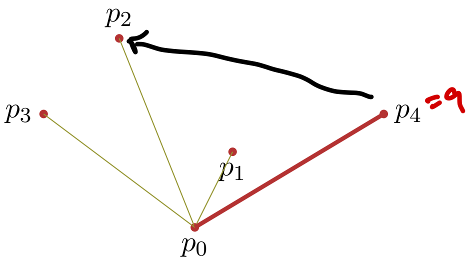


Illustration Jarvis



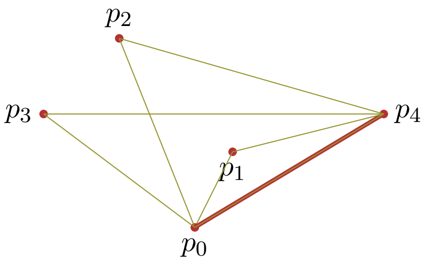
1. Set $\boxed{H \to \emptyset.}$
2. Find the lowest point q.
3. Find the rightmost point p, from q's point of view

Illustration Jarvis



1. Set $H \to \emptyset$.
2. Find the lowest point q.
3. Find the rightmost point p, from q's point of view
4. Add p to H.

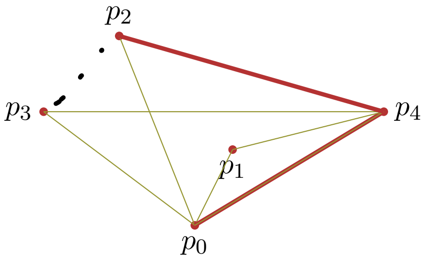
Illustration Jarvis



1. Set $H \to \emptyset$.
2. Find the lowest point q.
3. Find the rightmost point p, from q's point of view
4. Add p to H.
5. Set $q \leftarrow p$ and repeat from step 3 until q is the lowest point

$\leftarrow$

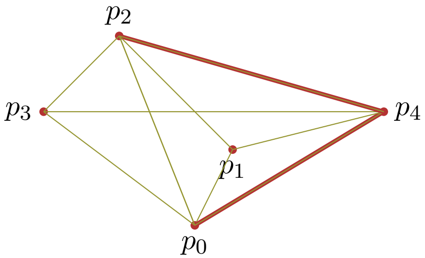
Illustration Jarvis



$H\{p_0, p_1, p_2, p_3\}$

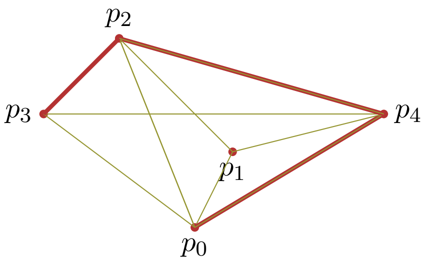
1. Set $H \to \emptyset$.
2. Find the lowest point q.
3. Find the rightmost point p, from q's point of view
4. Add p to H.
5. Set $q \leftarrow p$ and repeat from step 3 until q is the lowest point

Illustration Jarvis



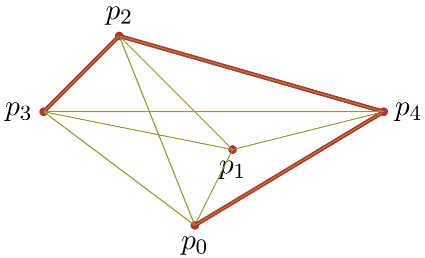
1. Set $H \rightarrow \emptyset$.
2. Find the lowest point q.
3. Find the rightmost point p, from q's point of view
4. Add p to H.
5. Set $q \leftarrow p$ and repeat from step 3 until q is the lowest point

Illustration Jarvis



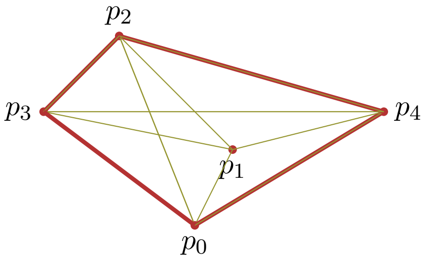
1. Set $H \rightarrow \emptyset$.
2. Find the lowest point q.
3. Find the rightmost point p, from q's point of view
4. Add p to H.
5. Set $q \leftarrow p$ and repeat from step 3 until q is the lowest point

Illustration Jarvis



1. Set $H \rightarrow \emptyset$.
2. Find the lowest point q.
3. Find the rightmost point p, from q's point of view
4. Add p to H.
5. Set $q \leftarrow p$ and repeat from step 3 until q is the lowest point

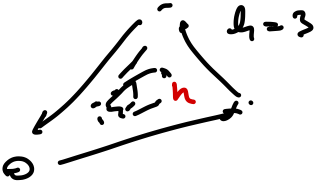
Illustration Jarvis



1. Set $H \to \emptyset$.
2. Find the lowest point q.
3. Find the rightmost point p, from q's point of view
4. Add p to H.
5. Set $q \leftarrow p$ and repeat from step 3 until q is the lowest point
6. H is the convex hull.

Graham Scan

- Graham Scan: Another algorithm that computes the convex hull
- See the implementation in the lecture slides
- Time complexity:
  - Jarvis March: $\mathcal{O}(h \cdot n)$ where h is the number of corner points on the convex hull
  - Graham Scan: $\mathcal{O}(n \log n)$
- **Question**: When does Jarvis March perform better?



$\log(n) \gg h = 3$

Graham Scan

- Graham Scan: Another algorithm that computes the convex hull
- See the implementation in the lecture slides
- Time complexity:
  - Jarvis March: $\mathcal{O}(h \cdot n)$ where h is the number of corner points on the convex hull
  - Graham Scan: $\mathcal{O}(n \log n)$
- **Question**: When does Jarvis March perform better?
- **Answer**: Jarvis March is better when h is small compared to n, as its time complexity depends on the number of corner points on the convex hull.
- Comment: Chan's algorithm improves on both, but is not taught in this course.

7. Past Exam Questions

Past Exam 2020: Task 1d)

(d) Sie haben einen sehr grossen Datensatz aus n unterschiedlichen Zahlen und wollen das k-kleinste Element finden ($k << n$). Wie machen Sie das effizient? Benennen Sie verwendete Datenstrukturen und Laufzeit des Algorithmus.

You have a huge data set with n different numbers, and you want to find the k-smallest element ($k << n$). How do you do this efficiently? Provide the used data structures and the runtime of the algorithm.

Past Exam 2020: Task 1d) – Solution

(d) Sie haben einen sehr grossen Datensatz aus n unterschiedlichen Zahlen und wollen das k-kleinste Element finden ($k << n$). Wie machen Sie das effizient? Benennen Sie verwendete Datenstrukturen und Laufzeit des Algorithmus.

You have a huge data set with n different numbers, and you want to find the k-smallest element ($k << n$). How do you do this efficiently? Provide the used data structures and the runtime of the algorithm.

We use a Max-Heap with k elements. The first k elements are filled in. For every following element, we check if it is smaller than the root. If it is smaller than the root, we replace the root by the new element and let it sink (heapify). At the end, the root contains the k-smallest element.
Runtime $n \log k$.
Alternative: use a linked list or array sorted decreasingly with with k elements and insert each new element with a step of bubble sort. Throw away new element. Runtime $n \cdot k$
Better alternative: use quick select with Blum's algorithm for the median selection: runtime n.

Past Exam 2020: Task 1e)

Für die folgenden ja/nein Fragen geben Sie jeweils die Antwort mit kurzer Begründung. Fassen Sie sich kurz!

For the following yes/no questions provide the answer with a brief explanation. Be brief!

(e) Die asymptotische Laufzeit eines randomisierten Algorithmus ist bis auf eine Konstante im schlechtesten Fall gleich gross wie im Erwartungwert.

The worst-case asymptotic running time and expected asymptotic running time are equal to within constant factors for any randomized algorithm.

◯ richtig/*correct* ◯ falsch/*wrong*

Begründung / *Justification*

Past Exam 2020: Task 1e) – Solution

Für die folgenden ja/nein Fragen geben Sie jeweils die Antwort mit kurzer Begründung. Fassen Sie sich kurz!

For the following yes/no questions provide the answer with a brief explanation. Be brief!

(e) Die asymptotische Laufzeit eines randomisierten Algorithmus ist bis auf eine Konstante im schlechtesten Fall gleich gross wie im Erwartungwert.

The worst-case asymptotic running time and expected asymptotic running time are equal to within constant factors for any randomized algorithm.

◯ richtig/*correct* $\surd$ falsch/*wrong*

Begründung / *Justification*

Quicksort hat z.B. erwartete Laufzeit von $\Theta(n \log n)$ aber im schlechtesten Falle $\Theta(n^2)$.

8. Outro

General Questions?

One more thing…

See you next time!

h_1 : 0011001

h_2 : 1100000

1111001

1111001

Have a nice week!