Algorithms: Sorting and Convex Hull

CS 002 - WI 2015
Friday, January 16, 2015

Today...

- Useful C++ features
- Sorting algorithms
- Convex-hull algorithms

Previously, on containers...

- We have arrays...
  - static arrays
    - random-access
    - contiguous memory (pointer-index equivalence)
    - fixed size (set at compile time)
    - limited size (limited to size of stack)
  - dynamic arrays
    - random-access
    - contiguous memory
    - fixed size (set at initialization)
    - can be "arbitrarily" large

Previously, on containers...

- Arrays are rather limiting!
  - no easy way to resize
  - pointers can be ugly to work with
  - no bounds checking!!
- Would like container that behaves more nicely
  - add / remove elements at will?
  - automatically resize as needed?
  - automatically keep track of sizes?

Standard Template Library: `std::vector<T>`

- Enter the std::vector
- Template class provided by STL
- Provides a resizable array
  - Can use the same array notation as normal
  - Can add or delete elements at end
  - Can resize at will

```
#include <vector>

std::vector<int> nums(20);
// array notation
for (int i = 0; i < 20; i++)
{
  nums[i] = i * 2;
}
// adding things to end
for (int j = 0; j < 20000; j++)
{
  nums.push_back(j + 42);
}
// resizing
nums.resize(50);
```

Standard Template Library: std::vector<T>

- Supports iteration with a special "iterator" class
- Request an iterator instance from a object method
- Iterators can be dereferenced *
- Iterators can be incremented ++

```
std::vector<int> nums;

...

std::vector<int>::iterator i;

for (i  = nums.begin();
     i != nums.end();
     i++)
{
    int foo = *i;
    printf("%d\n", foo);
}
```

Command-line arguments

- Recall that main() takes arguments
- These arguments are populated from the command line
- **`argc`**: the number of command line arguments 'plus 1'
- **`argv`**: the command line arguments as array
  - **`argv[0]`** is always the program name

```
#include <cstdio>
#include <cstdlib>

int main(int argc, char ** argv)
{
  if (argc != 2)
  {
    printf("usage: %s n\n",
            argv[0]);
  }
  else
  {
    printf("%f\n", atof(argv[1]));
  }
}
```

File I/O (C Standard Library)

- `fopen(filename, mode)`
  - returns a "`FILE *`"
  - use for subsequent calls
- `fread(buf, size, count, file)`
  - reads data into a buffer
- `fscanf(file, format, ...)`
  - reads formatted data, just like scanf()
- `fwrite(buf, size, count, file)`
  - writes data from a buffer
- `fprintf(file, format, ...)`
  - writes formatted data, just like printf()
- `fclose(file)`
  - closes the file handle

```
#include <cstdio>

char data[4096];
FILE * f = fopen("fish", "rb+");
if (!f)
{
    printf("fail\n");
}
else
{
    int num_read = fread(
      data, 1, 4096, f
    ); // reads up to 4KB from file
    fwrite(
      data, 1, num_read, stdout
    ); // prints data to terminal
}
```

File I/O (C++ Streams)

- `fstream`: C++ file stream
- `<<` / `>>`
  - 'formatted' write / read
- `fstream::getline(buf)`
  - read a single line into buffer
- `getline(file, string)`
  - read a single line into C++-style string
- `fstream::read(buf, size)`
  - reads into a block of memory
- `fstream::write(buf, size)`
  - writes from a block of memory
- `fstream::close()`
  - closes the file

```
#include <fstream>
#include <iostream>
#include <string>

std::string line;
std::fstream f;
f.open("fish");
if (f.is_open())
{
  while (std::getline(f, line))
  {
    std::cout << line << std::endl;
  }
  f << "protons!!";
  f.close();
}
```

Sorting!

Given an array of numbers...

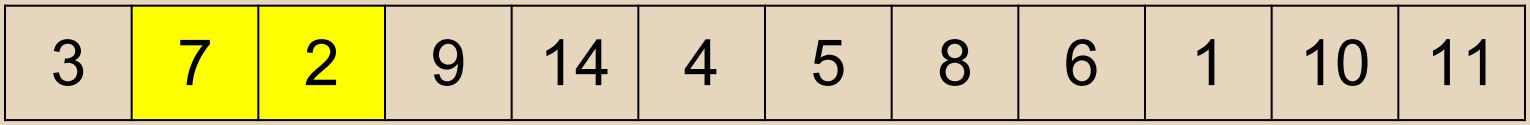
3	7	2	9	14	4	5	8	6	1	10	11

how do we sort them?

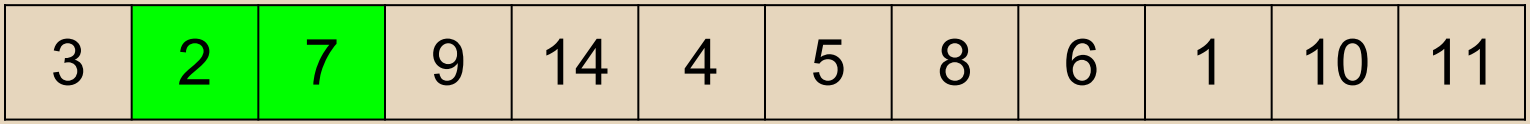
1	2	3	4	5	6	7	8	9	10	11	14

Sorting!

One idea: given two adjacent elements out of order...



swap them:



This is the bubble-sort principle.

Sorting!

Another idea: cut the array in half...

3	7	2	9	14	4

5	8	6	1	10	11

Sorting!

Another idea: cut the array in half...

3	7	2	9	14	4

5	8	6	1	10	11

sort the halves...

2	3	4	7	9	14

1	5	6	8	10	11

Sorting!

Another idea: cut the array in half...

3	7	2	9	14	4

5	8	6	1	10	11

sort the halves...

2	3	4	7	9	14

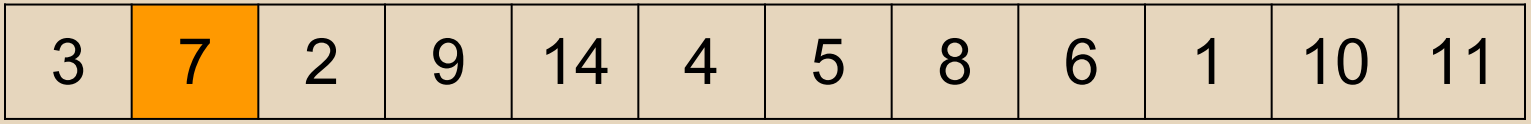
1	5	6	8	10	11

and merge the sorted halves.

1	2	3	4	5	6	7	8	9	10	11	14

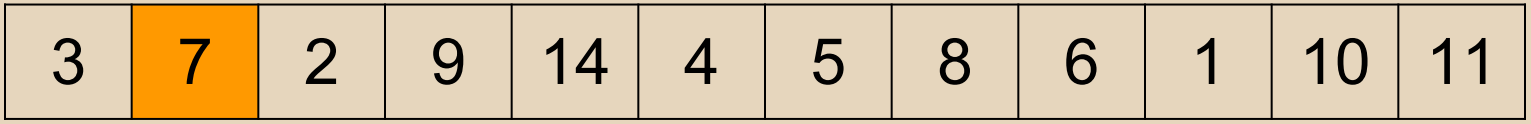
Sorting!

Another idea: choose some "pivot"...

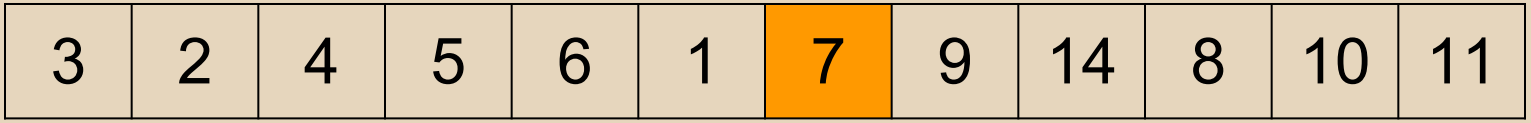


Sorting!

Another idea: choose some "pivot"...

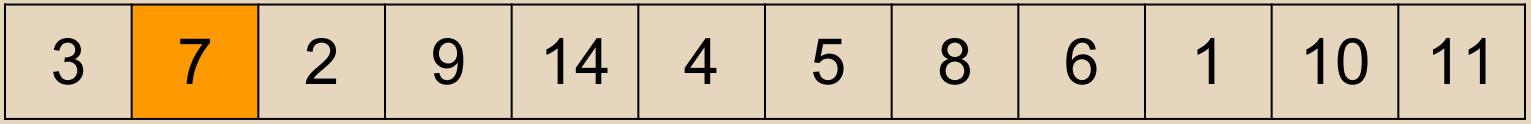


quickly put the pivot in the right place...

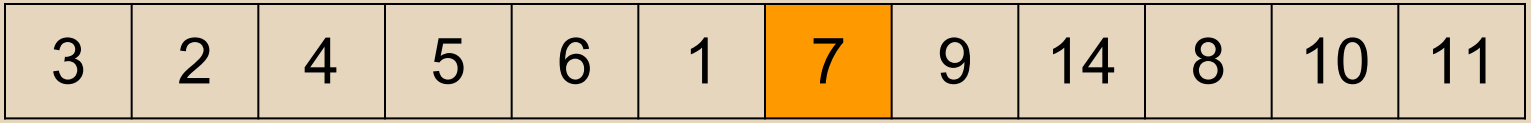


Sorting!

Another idea: choose some "pivot"...



quickly put the pivot in the right place...

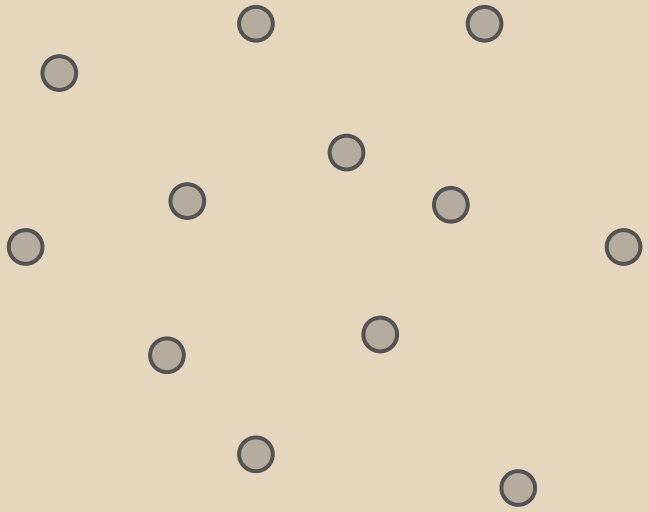


then repeat on the left and right half.

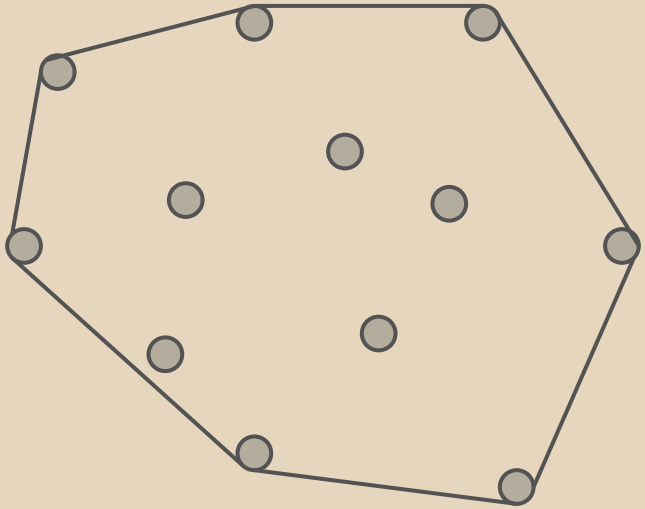
This is the quicksort algorithm.

Convex hull!?

Given some points...

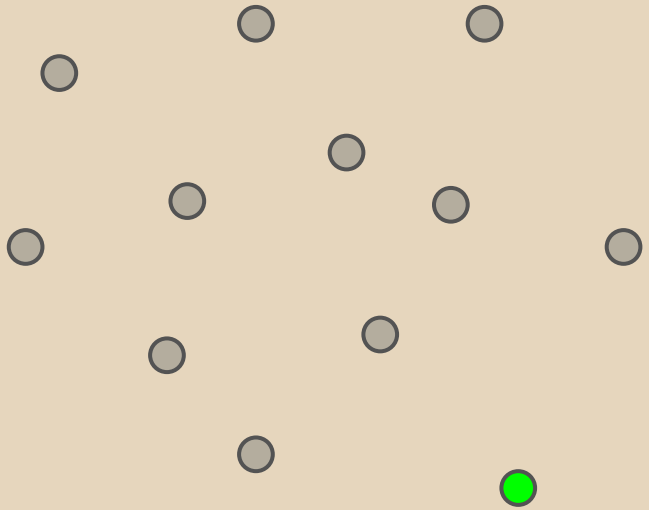


find the smallest convex polygon containing them.

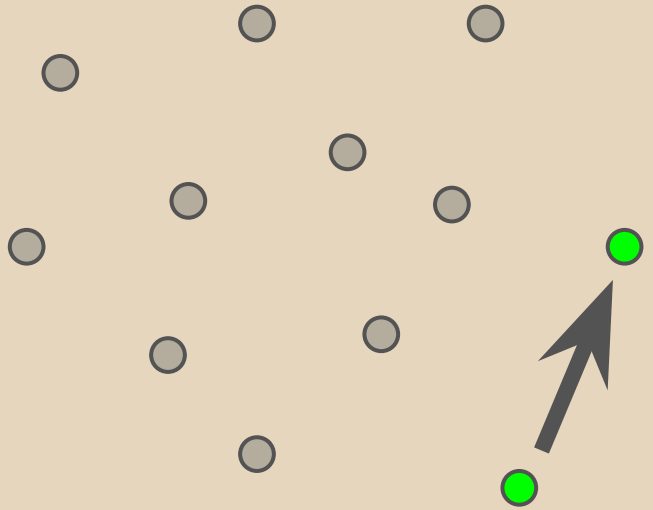


Convex hull - gift wrapping

We could start from the lowest point...

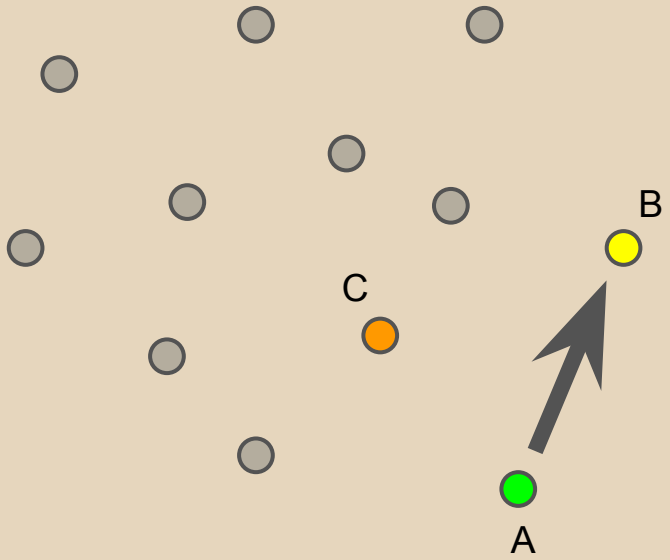


and find points such that all other points are to the 'left'.



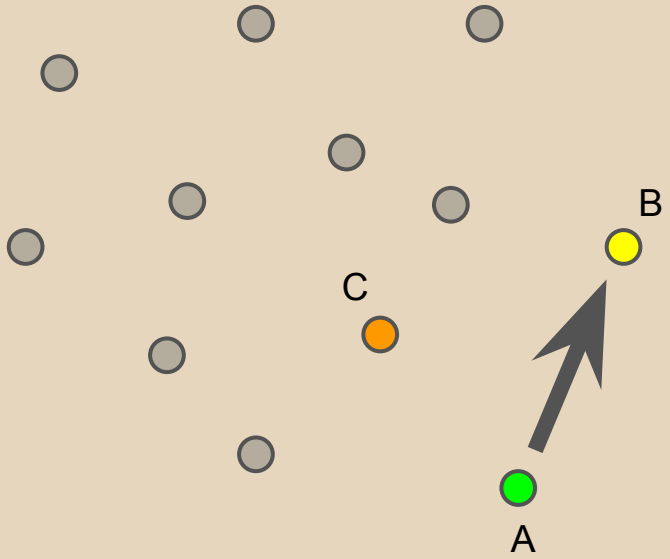
Convex hull - gift wrapping

How to tell if a point
is to the 'left'?

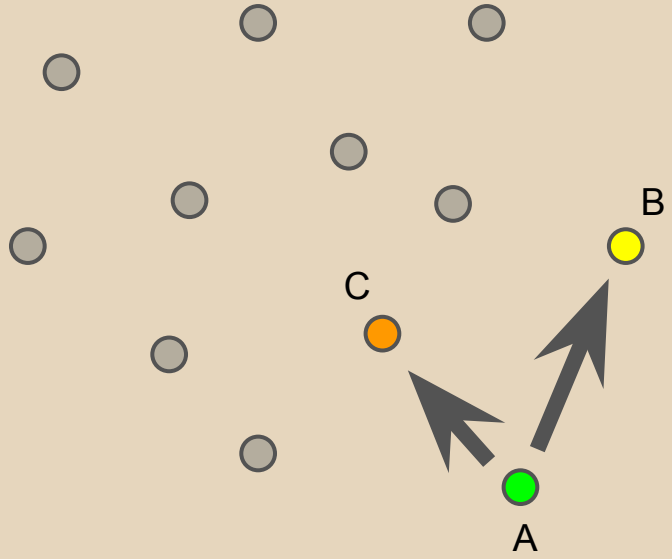


Convex hull - gift wrapping

A point C is to the 'left' of test line AB...

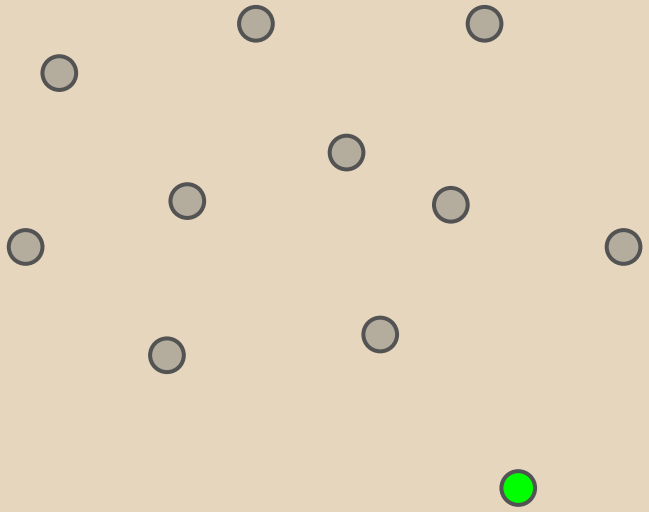


if AB × AC points 'out of the screen'.

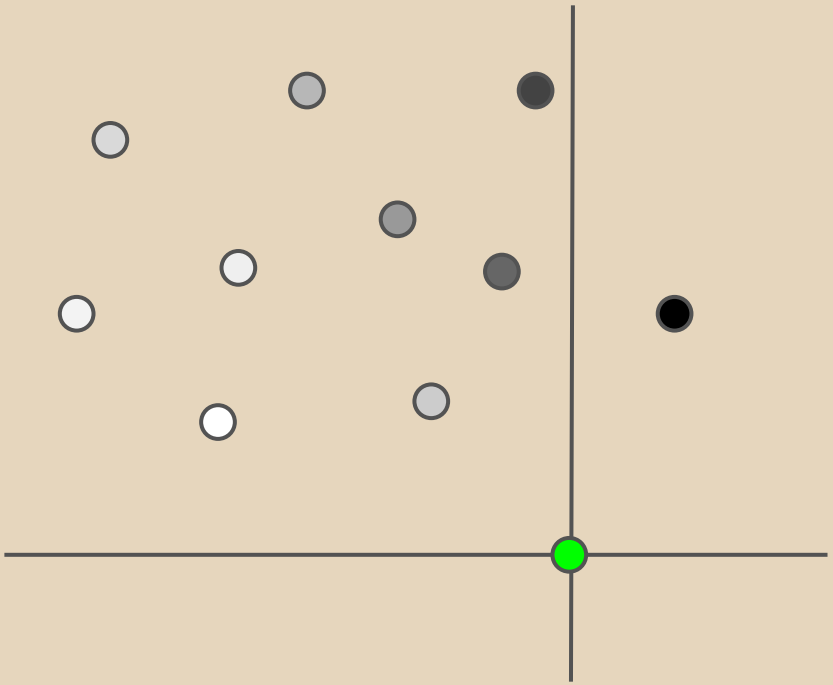


Convex hull - Graham scan

We could start from the lowest point...

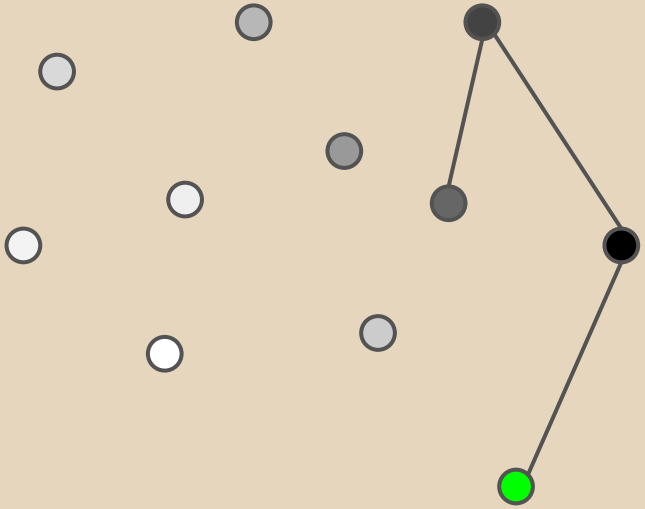


and sort all the points by "angle".

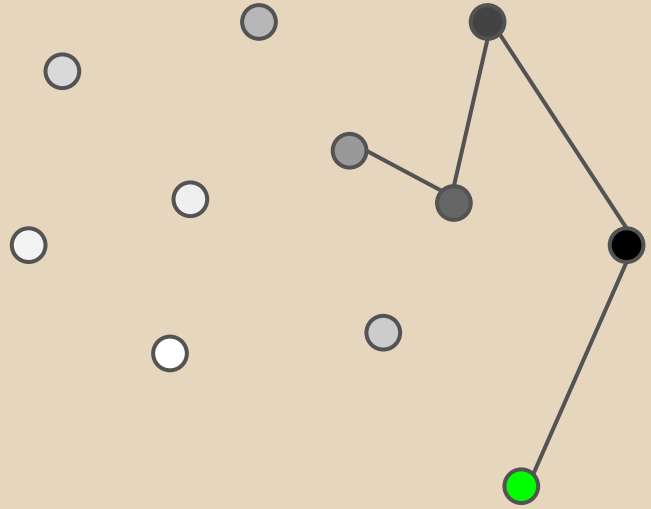


Convex hull - Graham scan

Traverse points in angle order. Assume they're in the hull...

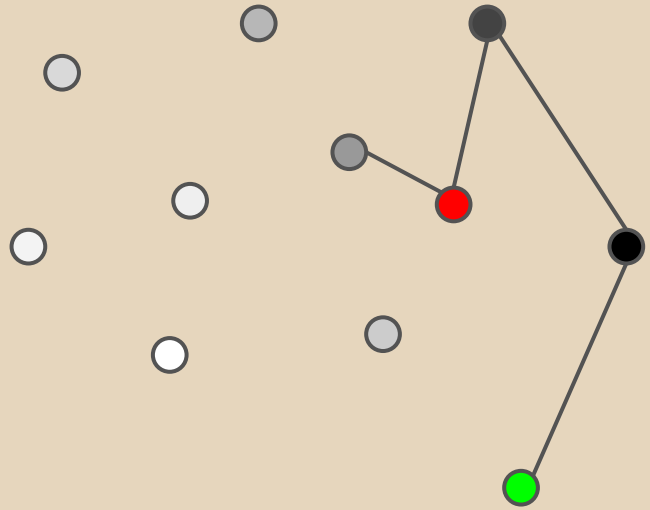


until we run into a 'right turn'.



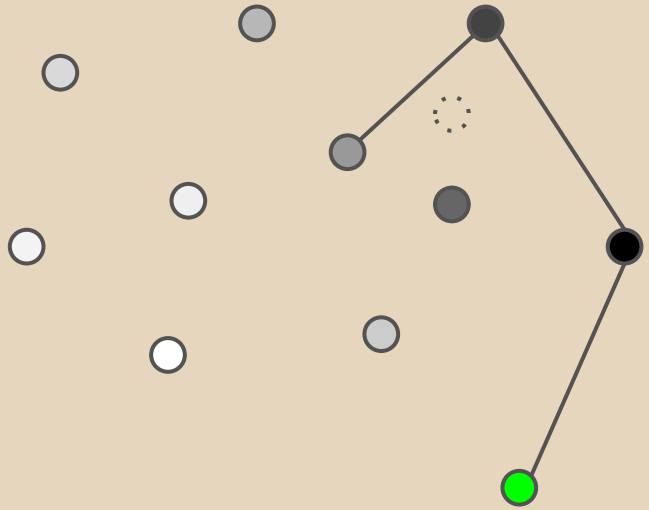
Convex hull - Graham scan

Remove points immediately before the one we just added from the hull...



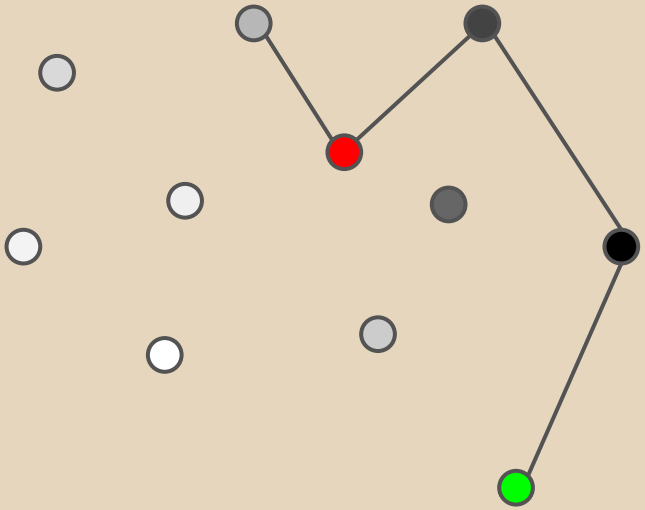
until no more right turns exist.

Depending on the layout, you may have to remove several points!

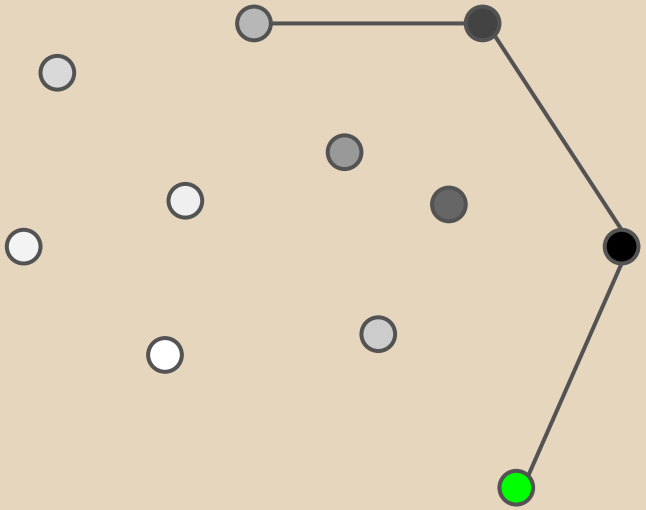


Convex hull - Graham scan

Then keep going, and going...

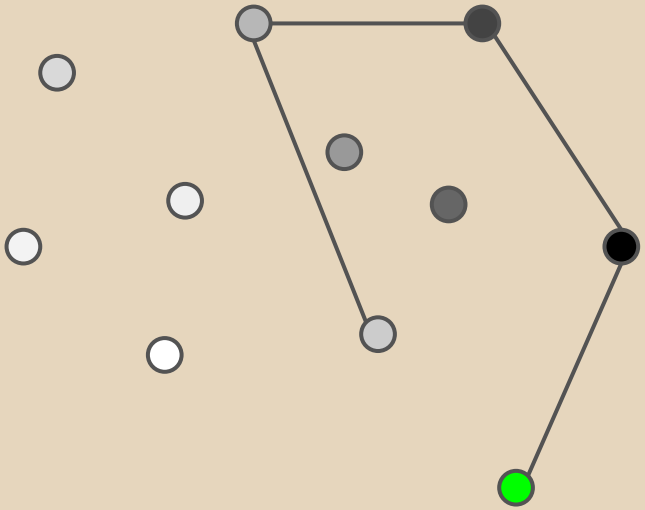


and going and going...

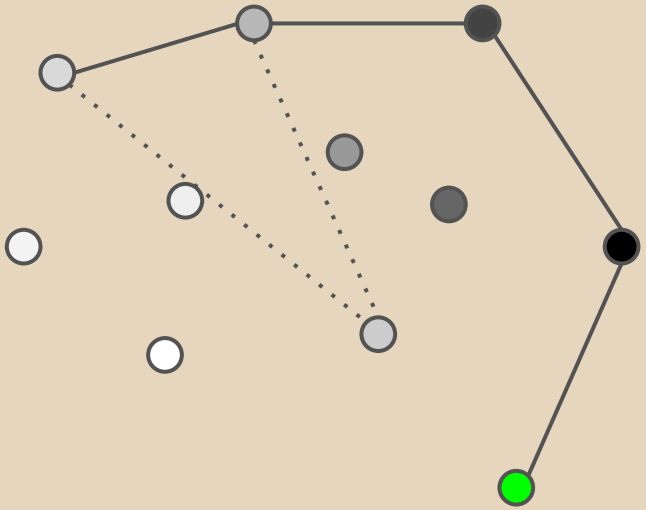


Convex hull - Graham scan

and going and going...

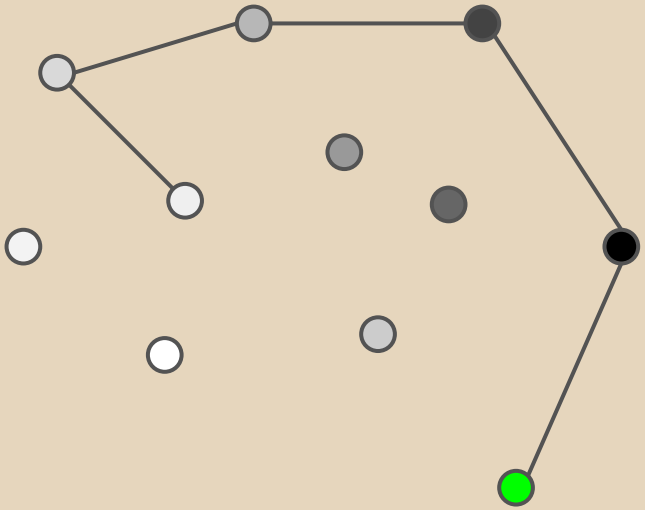


and going and going...

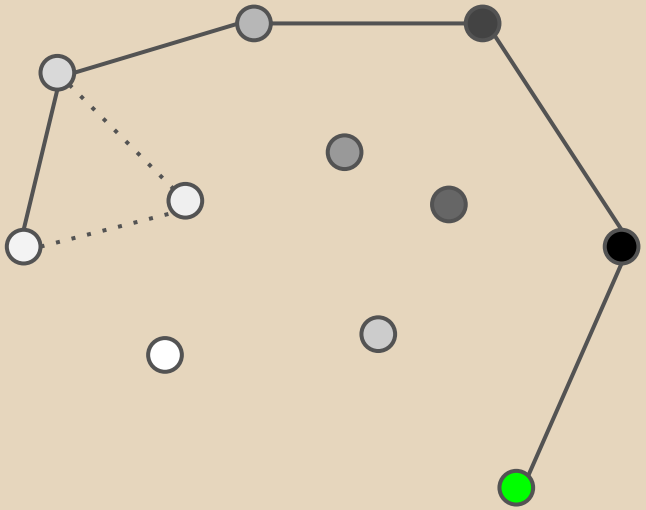


Convex hull - Graham scan

and going and going...

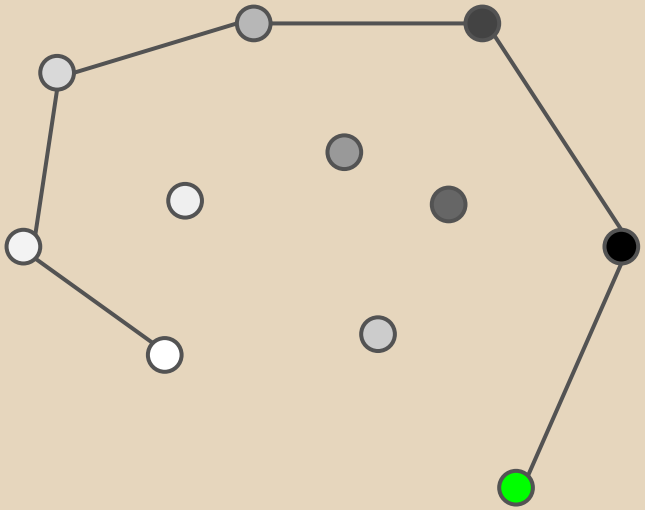


and going and going...

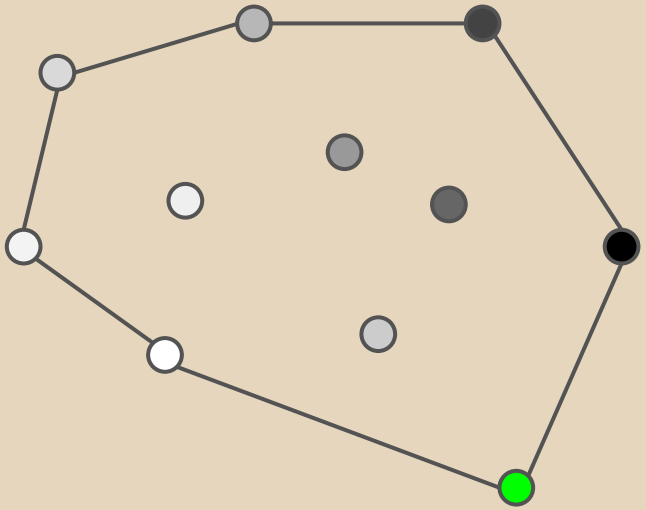


Convex hull - Graham scan

and going and going...



until you have a hull!



Demo time!!!

Extra Slides

The `auto` keyword (C++11)

- C++ is a statically typed language.
- Modern C++ compilers can do limited type inference.
  - Type still fixed at compile time -
  - but the compiler figures out the right type!
- This is the **`auto`** keyword.

```
std::vector<int> nums;

for(auto i  = nums.begin();
        i != nums.end();
        i++)
// i has type ...::iterator
// inferred by compiler
{
    printf("%d\n", *i);
}
```

The `auto` keyword (C++11)

- **`auto`** is capable of inferring types based on the type of the *variable initializer*.
- Variable must be explicitly initialized right away!
  - otherwise compiler has no idea what type you really want
- Use **`auto`** sparingly
  - overuse can result in confusion!

```
std::vector<int> nums;

for(auto i  = nums.begin();
        i != nums.end();
        i++)
// i has type ...::iterator
// inferred by compiler
{
    auto j; // not allowed
    auto j = *i; // OK
    // j has type int
    printf("%d\n", j);
}
```