

PG5600 iOS programming

Lesson # 3

Reminder

Everything is on github <https://github.com/BeiningBogen/iOS-Kristiania>

Slack is good! PG56000.slack.com

Review

- Functions
- Closures
- Enumeration
- Classes & Structs
- Stored Properties
- Methods
- Access control

Today - Swift (cont'd)

- Subscripts, Constructors and Inheritance
- ARC
- Optionals & Optional chaining
- Guard
- Type casting & Nested types
- Protocols
- Extensions
- Generics

Subscripts

- Shortcuts to retrieve and insert items in a collection, list, or sequence
- Set and get in the same way
- Can be defined in classes, structures and enums

Subscripts (Cont'd)

```
// Dictionary structures implement subscripts  
var studentsBySubject = ["ios": 10000, "android": 90, "wp": 10]  
  
// Access and set items using the key  
print(studentsBySubject["ios"]) // 10000  
studentsBySubject["ios"] = 500000
```

Subscripts (Cont'd)

- As calculated properties, they can be read-write or read only

```
class ExampleSubscript {  
    subscript(<parameters>) -> <returnType> {  
        // Getter is mandatory  
        get {  
            <statements>  
        }  
        // Setter is optional  
        set(<setterName>) {  
            <statements>  
        }  
    }  
}
```

Subscript overloading

- Define as many subscript as you want
- Swift is smart at guessing which to use (based on type)

```
class ExampleSubscript {  
    //...  
  
    subscript(pattern: String) -> Bool {  
  
    }  
  
    subscript(willBeDone: Bool) -> String {  
  
    }  
  
    //...  
}
```


Constructors

- Requires the use of named parameters
- Same as methods, they are bypassed using `_`, but it is not recommended
- Properties can be set in the constructor

```
class LivingThing {  
    let birth: Date  
  
    init(birth: Date) {  
        self.birth = birth  
    }  
}  
  
var aThing = LivingThing(birth: Date())
```

Constructors (Cont'd)

- Optionals and values with default value must not be set in the constructor

```
class LivingThing {  
    let birth: Date  
    var death: Date?  
    var isAlive: Bool = true  
  
    init(birth: Date) {  
        self.birth = birth  
    }  
}
```

```
var livingThing = LivingThing(birth: NSDate())
```

Constructors (Cont'd)

- You can have several constructors and they can call each other
- There are two different types of constructor:

Constructors (Cont'd)

Designated

- Primary constructor that must initialise all non-optional, non-initialised properties
- Must call it's superclass (by inheritance)
- There are often very few or just one **Designated** constructor
- All classes must have at least one, unless you have default values on all properties

Constructors (Cont'd)

Convenience

- Typically sets up a given state for the class
- Often requires fewer parameters
- Use them as a shortcut to set up a frequently used state
- **Convenience** constructors must first call a **Designated** constructor

Constructors (Cont'd)

```
class LivingThing {
    let birth: Date
    var death: Date?
    var isAlive: Bool = true

    init(birth: Date) {
        self.birth = birth
    }

    convenience init() {
        self.init(birth: Date())
        self.isAlive = false // has to be after self.init
    }
}

var livingThing = LivingThing(birth: Date())

// convenience
var livingThing2 = LivingThing()
```

Inheritance

A class can inherit

- Methods
- Properties

and everything else from another class

Inheritance (Cont'd)

- A class that inherits from another is called **subclass**
- The class that **subclass** inherits from is called **superclass**
- A class that does not inherit from anyone is called **base class**
- A **subclass** can call methods, properties and subscripts on **superclass**
- **subclass** can override a **superclass's** methods, properties and subscripts

Inheritance (Cont'd)

// Base classes and superclasses

```
class LivingThing {
    let birth: Date
    var death: Date?

    // Cannot be overwritten
    final var isAlive: Bool {
        return self.death == nil
    }

    init(birth: Date) {
        self.birth = birth
    }

    var description: String {
        return "I am a living thing that was born \(self.birth)"
    }
}
```

Inheritance (Cont'd)

// subclasses and superclasses

```
class Person : LivingThing {
    let firstName: String
    let lastName: String

    var fullName: String {
        return "\(self.firstName) \(self.lastName)"
    }

    // required - the subclass needs to implement the constructor
    required init(firstName: String, lastName:String, birth: Date) {
        self.firstName = firstName
        self.lastName = lastName
        // super can be used to call methods, properties and subscripts
        super.init(birth:birth)
    }

    func sayHello() -> String {
        return "Hello"
    }
}
```

Inheritance (Cont'd)

```
// subclass

class Student : Person {
    // Compile error because it's required
    init {

    }

    override var description: String {
        return "A student at Kristiania with the name \$(self.fullName)"
    }

    override fun sayHello() -> String {
        return "Halla \$(firstName)"
    }

    // Compile error
    override var isAlive: Bool {
        return true
    }
}

var student = Student(firstName: "John", lastName: "Doe", birth: Date())
student.firstName // John
student.description // A student at Kristiania with the name John

student.birth
```

ARC

- Usually, ARC automatically handles memory for you, but sometimes you have to do a bit yourself
- Implicit strong reference
- Anything that has a reference is kept in memory

Optional Chaining

```
if let street = kristiania.students.first?.address?.street {  
    print("the student lives on \(street).")  
} else {  
    print("Couldn't get the street name")  
}
```

You can

- Access properties
- Call methods
- Call subscripts

Type Casting

is

- Used to check the type of an instance

as

- Used to treat an instance as if it were another type in its type tree

Type Casting (Cont'd)

```
class LivingThing {}  
class Person: LivingThing {}  
class Animal: LivingThing {}
```

```
let living = [  
    Person(birth: NSDate()),  
    Animal(birth: NSDate()),  
    Person(birth: NSDate()),  
    Animal(birth: NSDate()),  
    Animal(birth: NSDate())  
]
```

```
living[0] is Person // true  
living[1] is Animal // true  
living[2] is Animal // false
```

as?

```
for item in living {  
    if let person = item as? Person {  
        print("Is alive: \(person.isAlive)")  
    } else if let animal = item as? Animal {  
        print("\(animal.roar())")  
    }  
}
```


Any & AnyObject

- AnyObject can represent an instance of any class type
- Any can represent an instance of any type, including feature types
- Should only be used when you actually need it, be explicit

Any & AnyObject (Cont'd)

// Cocoa-APIs & array can contain Any

```
let someObjects: [AnyObject] = [  
    Person(birth: NSDate()),  
    Person(birth: NSDate()),  
    Person(birth: NSDate())  
]  
  
for person in someObjects as [Person] {  
    print("Is alive: \(person.isAlive)")  
}
```

Any & AnyObject (Cont'd)

```
var things = [Any]()

things.append(0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))

for thing in things {
    switch thing {
    case 0 as Int:
        print("It was an int which was 0")
    case let someInt as Int:
        print("Found an Int that is: \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("A positive Double \(someDouble)")
    case is Double:
        print("Found some Double")
    case let someString as String:
        print("Found a string containing \"\(someString)\"")
    case let (x, y) as (Double, Double):
        print("a (x, y) with the values x: \(x), y: \(y)")
    default:
        print("Something else")
    }
}
```

Guard

```
let rocketDictionary : [String : String]? = [ "name" : "Falcon 9", "fuelName" : "liquid oxygen" ]

var factoryRobotsReady = true

func generateRocket(rocketDictionary: [String : String]?) {
    if factoryRobotsReady {
        if let actualDictionary = rocketDictionary {
            if let rocketName = actualDictionary["name"] {
                if let fuelName = actualDictionary["fuelName"] {
                    Rocket(rocketName, fuelName: fuelName)
                }
            }
        }
    }
}
```

Guard (Cont'd)

```
func generateRocketSchematics(rocketDictionary : [String : String]?) {  
    guard factoryRobotsReady else {  
        print("robots not ready")  
        return  
    }  
  
    guard let actualDictionary = rocketDictionary else {  
        print("no data to generate schematics")  
        return  
    }  
  
    guard let rocketName = actualDictionary["name"] else {  
        print("no rocket name")  
        return  
    }  
  
    guard let fuelName = actualDictionary["fuelName"] else {  
        print("no fuel name")  
        return  
    }  
  
    Rocket(rocketName, fuelName: fuelName)  
}
```

Extensions

- Extend functionality for a specific type
- Normal and static calculated properties
- Define new instance methods and class methods
- New init methods
- New subscripts
- Define a new nested type
- Allows you to implement a protocol for an existing type

Extensions (Cont'd)

```
extension String {  
    var uppercase: String { return self.toUpperCaseString }  
}
```

```
var name = "John Doe"  
name.uppercase // "JOHN DOE"
```

Protocols

- Same as interface in Java and other languages
- Defines a set of methods, properties, class methods, operators, and subscripts that fit a particular functionality
- Contains no implementation code

Protocols (Cont'd)

```
protocol LivingThing {  
    var mustBeSettable: Int { get set }  
    var doesNotNeedToBeSettable: Int { get }  
  
    class func someTypeMethod()  
    func random() -> Double  
    mutating func toggle() // Makes it possible to change properties  
}
```

Protocols (Cont'd)

- A protocol can be used anywhere that a type could be used
- A class, struct, or enum can implement multiple protocols
- Protocols can inherit from each other
- More about protocols when we switch to iOS

Protocol extensions

- You can create implementation code for methods
- Apple introduced Protocol oriented development

```
extension LivingThing {  
    func random() -> Double {  
        return 42  
    }  
}
```

Generics

- Many of Swift's standard library is built with generic code
- For example: Array and Dictionary are of the generic collections type

Generic functions

```
func printSequence<T: Sequence>(sequence: T) {  
    for part in sequence {  
        print(part)  
    }  
}
```

```
printSequence(sequence: "ABCDEF")  
printSequence(sequence: ["Aa", "Bb"])  
printSequence(sequence: ["A": "B", "B": "A"])
```

Generic Types

- Enums, structs, and classes can also be generic
- The Array and Dictionary are examples of generic structs

Generic Classes

```
class GenericClass<T> {  
    var object: T  
  
    init(object: T) {  
        self.object = object  
    }  
  
    func getObject() -> T {  
        return self.object;  
    }  
  
    func prinObject() {  
        print("Type of T is \$(self.object)");  
    }  
}  
  
var a = GenericClass<Int>(object: 1)  
a.prinObject()
```

Associated Types

- In a protocol one can create an alias (associated type) where it is up to the implementation to define the actual type
- You can refer to the type in methods and subscripts without determining the type in the protocol

Associated Types (Cont'd)

```
protocol Container {  
    typealias ItemType  
    mutating func append(item: ItemType)  
    var count: Int { get }  
    subscript(i: Int) -> ItemType { get }  
}
```

```
class Example: Container {  
    typealias ItemType = String  
    var array = [ItemType]()  
  
    func append(item: ItemType) {  
        self.array.append(item)  
    }  
  
    var count: Int {  
        get {  
            return array.count  
        }  
    }  
  
    subscript(i: Int) -> ItemType {  
        get {  
            return array[i]  
        }  
    }  
}
```

Operator overloading

```
func +(left: Balloon, right: Balloon) -> [Balloon] {  
    return [left, right]  
}
```

```
let balloon1 = Balloon()  
let balloon2 = Balloon()
```

```
let array = balloon1 + balloon2
```

Further reading

- See the TSPL book on the main topics of the lecture

The rest is not required for the course. Just for the exercises.

TSP

TSP (Travelling salesman problem) is a classic.

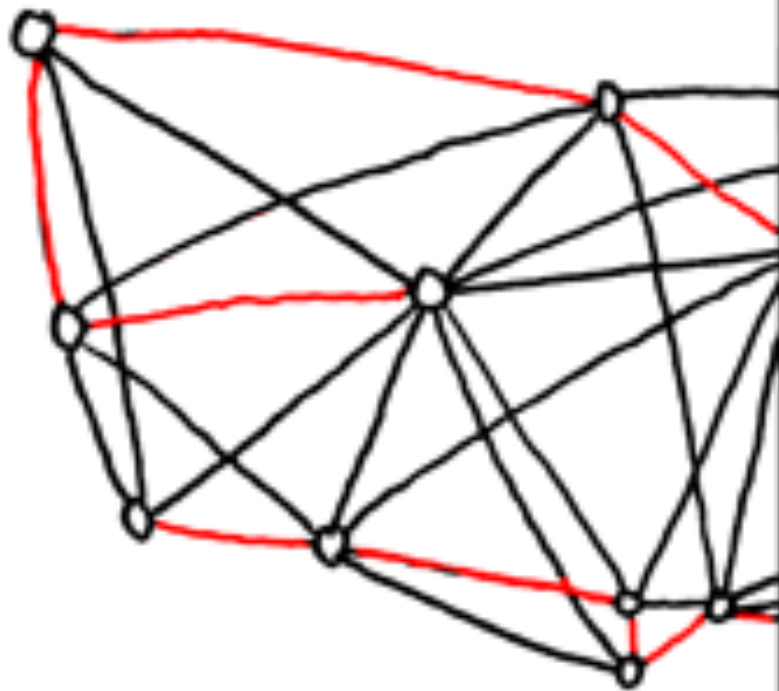
Complexity: $O(n!)$ 🤯

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

TSP (Cont'd)

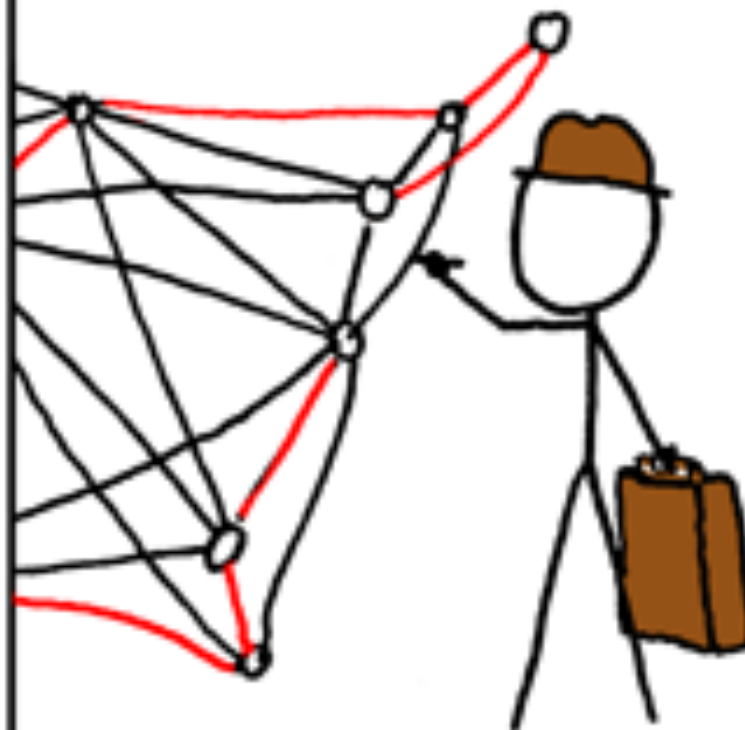
BRUTE-FORCE
SOLUTION:

$$O(n!)$$



DYNAMIC
PROGRAMMING
ALGORITHMS:

$$O(n^2 2^n)$$



SELLING ON EBAY:

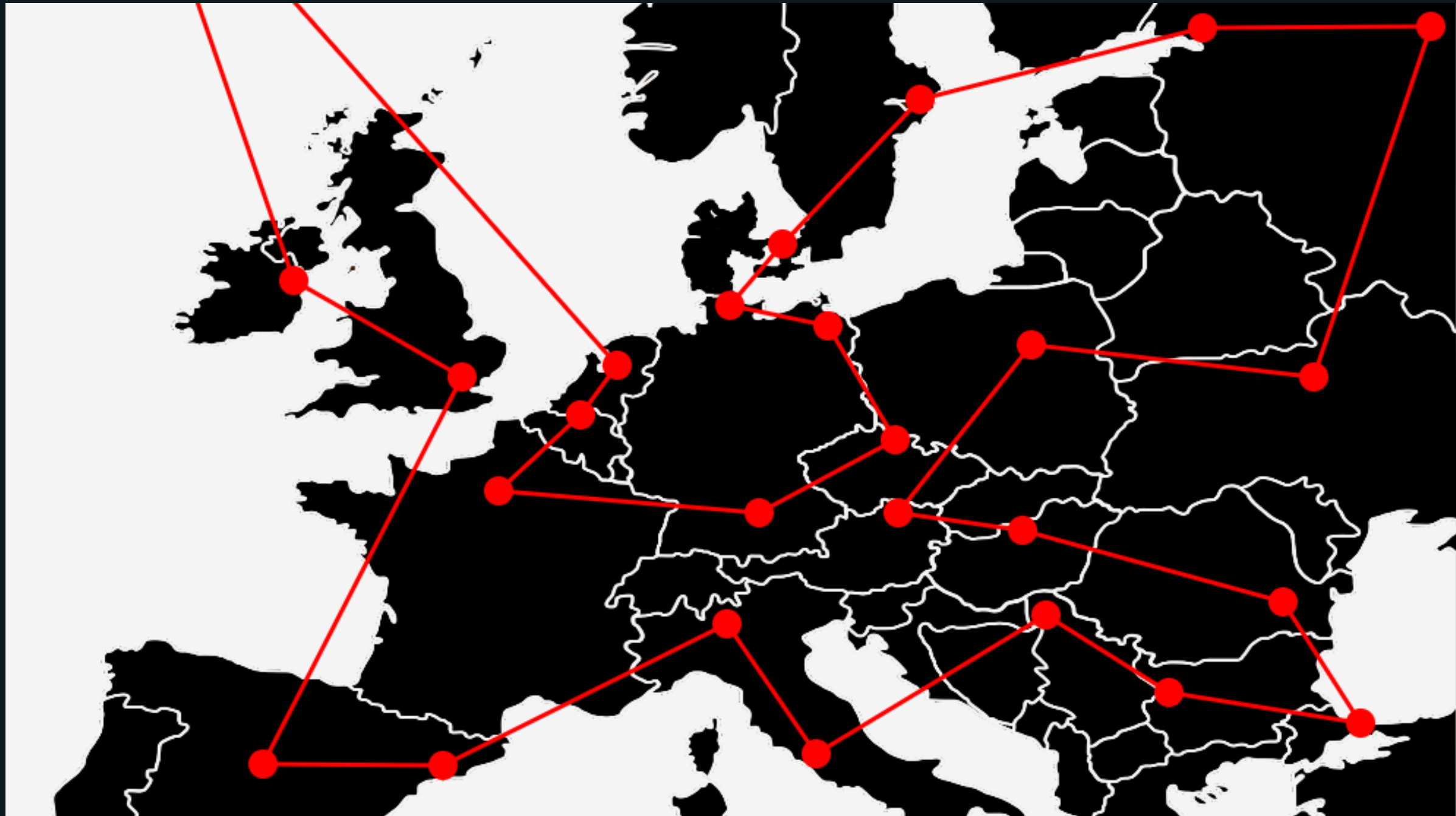
$$O(1)$$

STILL WORKING
ON YOUR ROUTE?

SHUT THE
HELL UP.



TSP (Cont'd)



TSP (Cont'd)

Rock band tour problem

A rockband wants to calculate the optimal route between cities.

TSP (Cont'd)

Australia: $7! = 5,050$ possible routes
1.4 hours (we can do the concert)

TSP (Cont'd)

Australia: $7! = 5,050$ possible routes

1.4 hours (we can do the concert)

USA: $22! = 1,124,000,727,777,607,680,000 = 1.1 \times 10^{21}$

3.6 years! (we need to plan ahead)

TSP (Cont'd)

Australia: $7! = 5,050$ possible routes
1.4 hours (we can do the concert)

USA: $22! = 1,124,000,727,777,607,680,000 = 1.1 \times 10^{21}$
3.6 years! (we need to plan ahead)

India: $29! = 8.8 \times 10^{30}$
 2×10^{11} years! (no tour in India)

TSP (Cont'd)

Australia: $7! = 5,050$ possible routes
1.4 hours (we can do the concert)

USA: $22! = 1,124,000,727,777,607,680,000 = 1.1 \times 10^{21}$
3.6 years! (we need to plan ahead)

India: $29! = 8.8 \times 10^{30}$
 2×10^{11} years! (no tour in India)

World: $100! = 9.3 \times 10^{157}$
 3×10^{138} years! (forget it!!!)

TSP (Cont'd)

Possible solution?

Use a heuristic for a "good" solution, not necessarily the best, but it's good enough.

Nearest Neighbour Algorithm

Greedy Algorithm: is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum

Nearest Neighbour Algorithm (Cont'd)

Given a list of points $[k]$, and a starting point x

- From x , find the closest point in $[k]$. Let's call this point y
- Remove y from $[k]$
- Add y to a queue
- y is the new x
- Repeat until $[k]$ is empty

Tasks

See Exercises on GitHub