CS3217 PS/Tutorial 1 Solution

Eric Han

January 21, 2022

Introduction

Eric Han

4th Year PhD Student

I am interested in scaling machine learning towards higher dimensions in Bayesian Optimization, Gaussian Processes, Convex and non-convex optimization and Reinforcement Learning.

Feel free to telegram me `@Eric_Vader`/on our group:

1. regarding the module,
2. research,
3. grad sch,
4. NUS
5. etc. . .

Expectations

Expectations of you

1. Fill the seats from the front.
2. Come prepared to the Tutorial.
   2.1 You will be asked at random to discuss your answer.
3. Refrain from taking pictures of the slides.
   3.1 Learn to take good notes.
   3.2 The slides will *not* be distributed.
4. Try to be on time.

Any comments or suggestions for the lessons going forward?

PS Question 1

How do you implement `Stack` and `Queue` efficiently?

- What is a `Stack`?
- What is a `Queue`?
- What are the differences between them?

PS Question 1 - Answer

Stack

Use a normal `Array` with the `append` and `popLast` methods.

Queue (value type semantics[1])

- Use two stacks, see, e.g. Cracking The Coding Interview.
- Use a linked list.
  - Danger of retain cycles causing memory leaks.
  - Must use reference type[1].
- Use an `ArraySlice`, which has an efficient `popFirst` method
- Use a dictionary `[Int: T]`.
  - To enqueue, insert to the dictionary with the largest number.
  - To dequeue, you pop the smallest number.

[1] Not penalized; will be covered next few weeks.

PS Question 2

What changes do you make in the skeleton code?

- What are some of the advantages of your changes?
- What are some of the disadvantages of your changes?

PS Question 2 - Answer

Possible Solution

Modify `Node` and `Edge` to be hashable instead of just equatable. Modify the type variable `T` to be hashable as well.

- One can use an adjacency list to back the graph, i.e. `[N: Set<E>]` or `[N: [E]]`.
- Violates the open-closed principle, as we are modifying the class (vs. extending).

Improved Solution

Create a separate class, e.g. `FastNode` and `FastEdge`, that adopts the changes, instead of changing `Node` and `Edge` directly.

PS Question 3

> *`checkRepresentation` is supposed to check if the representation invariants of the ADT is fulfilled or not.*

Where do you put `checkRepresentation` in your code?

- Did you discard it?
- If not where did you put it?

PS Question 3 - Answer

Invalid Reasons to discard `checkRepresentation`

- It slows down performance;
  - `assert` is turned off in production, see Swift Docs.
- Invariants are already satisfied;
  - need to make sure that the invariants in your specific implementation brings is fulfilled.
- Test cases are sufficient to prevent all bugs; this is an excuse.
  - When writing checkRepresentation, we are in a different frame of mind. This is doubly defensive and will help catch mistakes which unit tests might miss.

Valid Reasons to discard `checkRepresentation`

- `checkRepresentation` slows down testing; You should include benchmarks.
  - It should still be used as a final thorough test before deployment.

Recommended places to put `checkRepresentation`

- End of constructors.
- End of mutating methods.
- Start of mutating methods.
- End of non-mutating methods.

Places not to put `checkRepresentation`

- Start of constructors.
- Start of non-mutating methods.

PS Question 4

What graph representation did you choose to use?

- Why did you choose that?
- Are there other alternatives?

PS Question 4 - Answer

Two popular choices, with tradeoffs:

- Adjacency list:
  - Fast specifically for BFS/DFS,
  - Easy to implement.
  - Less space for sparse graphs.
- Adjacency matrix:
  - Less space for non-sparse/complete graphs.

PS Question 5

For the `Tree` ADT, did you extend your `Graph` ADT or did you write from scratch?

- Why did you extend?
- Or why not?

PS Question 5 - Answer

One possible solution is write from scratch (just 4 lines):

```
struct Tree<T> {
    var value: T
    var children: [Tree<T>]
}
```

PS Question 6

Discuss your experience with Test Driven Development on implementing your Graph ADT.

Tutorial Question 1

Let us create a function in Swift that reads a CSV file and deserialises it into an array of structs. Refactor this code to improve its coding style.

```
struct Person {
    let name: String
    let age: Int
}
```

```swift
func des(s: String) -> [Person] {
    var people = s.split(separator: "\n")
    var array: [Person] = []
    for p in people {
        let pSplit = p.split(separator: ",")
        array.append(Person(name: String(pSplit[0]), age: Int(pSplit[1])!))
    }

    return array
}
```

Tutorial Question 1 - Answer

```swift
func createPerson(fromRow row: [String]) -> Person? {
    let person = row.split(separator: ",")
    guard person.count == 2,
            let name = String(person[0]),
            let age = Int(person[1]) else {
        return nil
    }
    return Person(name: name, age: age)
}
```

```
/// This function ignores rows that represent invalid input.
func deserialise(csvString: String) -> [Person] {
    let rowData = s.split(separator: "\n")
    return rowData.compactMap { createPerson($0) }
}
```

Notes:

- Can also write it in the original imperative style instead of functionally.
- Can opt to null-coalesce the age variable (i.e. `age ?? 0`) instead of discarding.

Tutorial Question 2

There are several ways for functions in Swift to indicate that something bad has occured. For example, one can return a special value (e.g. `-1`), return `nil`, or throw an exception.

1. Why and when is one method preferred over the other?
2. For these scenarios, suggest which method you think should be used.
   - 2.1 Parsing JSON function, on receiving invalid input.
   - 2.2 A function takes in a file name and outputs the contents of the file. The function receives a file name, but the file does not exist.
   - 2.3 Adding an element to a set, but the element already exists in the set.
   - 2.4 Passing an unsorted array into a binary search function.

Tutorial Question 2 - Answer

Least to Most Preferable:

1. Special value is the least preferable as it is not communicative to the caller.
2. `nil` is preferable in these cases:
   - cases where the client of the code can benefit from the language features, e.g. optional chaining instead of doing a series of `try` statements
   - consistency with the language's features, e.g. failable initialisers
   - where it is semantically correct for it to be a null value
3. Exceptions in preferable in these cases:
   - when there is more than one way the code can "fail"
   - when there is benefit in giving additional information

Further discussion:

1. Opening a file, which technique should we use and why?
2. Adding an element to the set, which technique should we use and why?

Tutorial Question 3

Consider the code snippet below. Identify which comments are useful, and which ones are not so useful.

```
/// This function computes the square root of the number passed in.
/// Uses the Newton-Raphson method, set for 100 stages. It converges
/// for virtually all possible input.
///
/// - Parameter num: the number to be passed in
/// - Returns: The square root of the number
func squareRoot(_ num: Double) -> Double? {
    // Check if num is negative
    if (num < 0) {
        return nil // No square root
    }
```

```
    var result: Double = 1 // Initialise to 1
    for _ in 0..<100 { // Run 100 times
        result = (result + num / result) / 2 // One Newton-Raphson step
    }

    return result
}
```

Tutorial Question 3 - Answer

Suggested Answer

```
/// Uses the Newton-Raphson method, set for 100 stages. It converges
/// for 1e-22 to 1e22.
/// It returns `nil` for invalid input, i.e. negative numbers.
func squareRoot(_ num: Double) -> Double? {
    if (num < 0) {
        return nil
    }
```

```
var result: Double = 1
for _ in 0..<100 {
    // One Newton-Raphson step
    result = (result + num / result) / 2
}

return result} // Slide has no space
```

Tutorial Question 4

Many programming languages have different ways to parse strings to integers. Here are some of them:

- In Swift, one uses e.g. `Int("345")`, which returns an `Int?`
- In C++, one uses e.g. `std::stoi("345")`
- In Ruby, one uses e.g. `"345".to_i`

Discuss about these different approaches: when is one preferable than the other? Why did the language designer decide to implement one over the other? If you were to create your own struct and you wish to convert a string to your struct, which one do you prefer?

Tutorial Question 4 - Answer

The Swift way would be most preferable as it is the most descriptive:

1. `Int` is very clear.
2. `stoi` is unclear to people who have not used C++.
3. `to_i` can be misleading as it can also be `i` the variable.

When we create custom objects, we will have custom constructor - an initialiser for the object that takes in a string. Nice, simple and clean.

C++ has `std::stoi` to be consistent with C APIs such as `atoi` and `atoll`. Ruby is an expressive language, and having `to_i`, `to_s`, `to_f`, etc. allows the client of the code to chain things and write one-liners.