Nick Troccoli
CS 106X

Section #8 Solutions

Based on handouts by various current and past CS106B/X instructors and TAs.

1. Kruskal's Algorithm *(graphs)*

***Edges*:** G-A, G-H, I-C, A-B, B-C, G-D (these are undirected, so G-A is the same as A-G, for example)

***Cost*:** 26

2. Topological sort *(graphs)*

```
map: {heat:0, egg:0, measure dry:0, melt:0, oil:1, add wet:3, make:2, eat:1, clean:1}
queue: {heat, egg, measure dry, melt}      ordering: {}

dequeue heat, add heat to ordering, update in-degree of oil to 0, enqueue oil
map: {heat:0, egg:0, measure dry:0, melt:0, oil:0, add wet:3, make:2, eat:1, clean:1}
queue: {egg, measure dry, melt, oil}       ordering: {heat}

dequeue egg, add egg to ordering, update in-degree of "add wet" to 2
map: {heat:0, egg:0, measure dry:0, melt:0, oil:0, add wet:2, make:2, eat:1, clean:1}
queue: {measure dry, melt, oil}            ordering: {heat, egg}

dequeue "measure dry", add "measure dry" to ordering, update in-degree of "add wet" to 1
map: {heat:0, egg:0, measure dry:0, melt:0, oil:0, add wet:1, make:2, eat:1, clean:1}
queue: {melt, oil}                         ordering: {heat, egg, measure dry}

dequeue melt, add melt to ordering, update in-degree of "add wet" to 0, enqueue "add wet"
map: {heat:0, egg:0, measure dry:0, melt:0, oil:0, add wet:0, make:2, eat:1, clean:1}
queue: {oil, add wet}                      ordering: {heat, egg, measure dry, melt}

dequeue oil, add oil to ordering, update in-degree of make to 1
map: {heat:0, egg:0, measure dry:0, melt:0, oil:0, add wet:0, make:1, eat:1, clean:1}
queue: {add wet}                           ordering: {heat, egg, measure dry, melt, oil}

dequeue "add wet", add "add wet" to ordering, update in-degree of make to 0, enqueue make
map: {heat:0, egg:0, measure dry:0, melt:0, oil:0, add wet:0, make:0, eat:1, clean:1}
queue: {make}                        ordering: {heat, egg, measure dry, melt, oil, add wet}

dequeue make, add make to ordering, update in-degrees of make and clean to 0, enqueue both
map: {heat:0, egg:0, measure dry:0, melt:0, oil:0, add wet:0, make:0, eat:0, clean:0}
queue: {eat, clean}            ordering: {heat, egg, measure dry, melt, oil, add wet, make}

dequeue eat, add eat to ordering
map: {heat:0, egg:0, measure dry:0, melt:0, oil:0, add wet:0, make:0, eat:0, clean:0}
queue: {clean}            ordering: {heat, egg, measure dry, melt, oil, add wet, make, eat}

dequeue clean, add clean to ordering
map: {heat:0, egg:0, measure dry:0, melt:0, oil:0, add wet:0, make:0, eat:0, clean:0}
queue: {}
final ordering: {heat, egg, measure dry, melt, oil, add wet, make, eat, clean}
```

No, this is not the only valid topological sort ordering. There are many! Any valid ordering just lists every dependent node in the graph before all of its dependencies. A couple other valid orderings are:

```
{melt, egg, heat, measure dry, oil, add wet, make, eat, clean}  (re-order first 4)
{egg, measure dry, melt, add wet, heat, oil, make, eat, clean}  (list heat/oil later)
{heat, egg, measure dry, melt, oil, add wet, make, clean, eat}  (switch last 2)
```

3. Inheritance and Polymorphism Trace *(inheritance/polymorphism)*

```
Lettuce* var1 = new Bacon();
Bacon* var2 = new Mayo();
Lettuce* var3 = new Hamburger();
Bacon* var4 = new Hamburger();
Lettuce* var5 = new Lettuce();

var1->m1();                 cout << endl;   // L1 / L2 / B1
var1->m2();                 cout << endl;   // L2
var1->m3();                 cout << endl;   // COMPILER ERROR
var2->m1();                 cout << endl;   // L1 / H2 / L2 / B1
var2->m2();                 cout << endl;   // H2 / L2
var2->m3();                 cout << endl;   // M3 / L1 / H2 / L2 / B1
var2->m4();                 cout << endl;   // COMPILER ERROR
var3->m1();                 cout << endl;   // L1 / H2 / L2 / B1
var3->m2();                 cout << endl;   // H2 / L2
var4->m2();                 cout << endl;   // H2 / L2
var4->m3();                 cout << endl;   // B3
var4->m4();                 cout << endl;   // COMPILER ERROR
((Bacon*) var1)->m1();      cout << endl;   // L1 / L2 / B1      // (unchanged behavior)
((Bacon*) var1)->m3();      cout << endl;   // B3                // (cast makes it compile)
((Mayo*) var5)->m3();       cout << endl;   // CRASH             // (cast too far down)
((Lettuce*) var4)->m3();    cout << endl;   // COMPILER ERROR    // (Lettuce has no m3)
((Hamburger*) var2)->m4();  cout << endl;   // M4
((Mayo*) var2)->m4();       cout << endl;   // M4
```

4. Rigged Dice *(inheritance)*

```
// RiggedDice.h

#include "dice.h"
#pragma once

class RiggedDice : public Dice {
public:
    RiggedDice(int count, int min);
    virtual int getMin() const;
    virtual void roll(int index);
    virtual int total() const;
    virtual std::string toString() const;

private:
    int min;
};

// RiggedDice.cpp implementation shown on next page
```

```
// RiggedDice.cpp

#include "riggeddice.h"
using namespace std;

RiggedDice::RiggedDice(int count, int min) : Dice(count) {
    if (min < 1 || min > 6) throw min;
    this->min = min;
}

int RiggedDice::getMin() const {
    return min;
}

int RiggedDice::total() const {
    return Dice::total() + 1;
}

void RiggedDice::roll(int index) {
    // need to roll this die at least once
    do {
        Dice::roll(index);
    } while (getValue(index) < min);
}

string RiggedDice::toString() const {
    ostringstream out;
    out << "rigged ";
    out << Dice::toString();
    out << " min " << getMin();
    return out.str();
}
```