숙제 #5: 함수 재귀 Part 2

건국대학교 소프트웨어학과 | 2019 컴퓨테이션이론 | 담당: 차리서

이 문서는 5차 숙제를 위한 문제지 겸 설명서로서, 이 문서의 내용(과 수업 시간에 설명한 내용)만 알면 숙제를 해결할 수 있도록 구성되어 있습니다. (문제를 풀기 위해 다른 곳을 찾아봐야한다고 생각하기 전에 이 문서의 내용을 다시 한 번 충실히 읽고 이해하기를 추천합니다.) 단, 이번 5차 숙제는 4차 숙제에서 내용 상 다소 이어지기 때문에, 내용 이해를 위해서는 간혹 4차 숙제를 참고해야할 수는 있습니다. 물론, 어디까지나 내용 이해에 필요할 뿐이고, 4차 숙제에서 작성했던 함수들을 직접 사용해야하는 경우는 없습니다.

함께 첨부하는 `HW5.hs` 파일에 각 문제의 답에 해당하는 내용을 작성한 후, 완성된 `HW5.hs`만 숙제 제출 화면에 업로드하면 됩니다. 제출 기한은 11/26(화) 밤 10시까지입니다.

Contents

1 고정점 (fixpoint)

정의역과 공변역이 같은 어떤 단항 함수 f를 어떤 인자 p에 적용한 결과인 $f\ p$가 바로 그 인자 p 자신과 같다면 ($f\ p = p$), 이러한 값 p를 함수 f의 고정점 (fixed point 혹은 간단히 fixpoint) 혹은 부동점(invariant point)이라고 부릅니다.[1] 예를 들어, 인자로 주어진 수를 제곱하는 함수 (`^2`)는 인자 1에 적용할 경우 그 결과가 그대로 1이며:

$$(\texttt{\^{}2})\ 1 = 1 \qquad \text{즉,} \qquad 1^2 = 1$$

따라서 1은 함수 (`^2`)의 고정점입니다.

함수에 따라서 고정점은 전혀 없을 수도 있고, 하나만 있을 수도 있고, 여러 개 있을 수도 있습니다. 고정점이란 결국 "함수값(리턴값)이 인자와 동치인 지점"이기 때문에, 이를 만족시키는 "정의역 상의" 값들은 모두 고정점입니다. (정의역에서 벗어난 값들은 당연히 고정점이 아닙니다. 예를 들어, 실수에 대해 정의된

[1] 이 이름은 고정소수점 수(fixed-point numbers)나 부동소수점 수(floating-point numbers)와는 아무 관계 없습니다. 고정점이나 부동점의 "점"은 소수점이 아닙니다. 게다가, 부동점의 부동은 不動이고, floating-point numbers의 번역어인 부동소수점 수의 부동은 浮動입니다.

함수라면 위 조건을 만족시키는 실수만 고정점이고 허수는 위 조건을 만족시켜도 고정점이 아닙니다.) 다르게 말하자면, 해당 함수의 함수값을 y로 놓고 x-y 평면 상에 함수의 그래프를 그려보았을 때 해당 함수의 그래프가 $y = x$라는 기울기 1 (즉, 45도) 짜리 직선 그래프와 만나는 지점들은 모두 고정점입니다.

예를 들어, 만일 정의역이 실수인 함수 f의 정의가 (이해를 돕기 위해 여기서만 전통적인 수식 표기를 써서)

$$f(x) = x^2 - 3x - 5$$

라고 한다면, 결국 $f(p) = p$ 즉:

$$p^2 - 3p - 5 = p$$

를 만족시키는 p 값(들), 즉, 방정식

$$p^2 - 4p - 5 = 0$$

의 두 실근들인 5와 -1이 함수 f의 고정점들입니다. 다르게 말하자면, x-y 평면 상에서 아래 두 그래프:

$$y = x^2 - 3x - 5 \qquad \text{와} \qquad y = x$$

를 각각 그렸을 때 이들 두 그래프가 겹치는 두 지점의 x (혹은 y, 같으니까) 값들이 함수 f의 고정점입니다.

다른 다양한 경우들의 예를 들자면:

- 인자로 주어진 수를 1 증가시키는 함수 (`1+`)는 고정점이 전혀 없고 (이 함수의 그래프는 $y = x$ 그래프와 평행선이며, 결코 만나지 않습니다),
- 인자로 주어진 정수를 7로 나눈 몫을 구하는 함수 (`` `quot` 7 ``) 혹은 (`flip quot 7`)은 고정점이 딱 하나 (0) 있으며 (이 함수의 그래프는 정수들의 가로세로 격자 상에서 계단식으로 완만하게 (기울기 1/7) 증가하며, $y = x$ 그래프와는 원점에서만 만납니다),
- 인자로 수어진 수를 제곱하는 함수 (`^2`)는 고정점이 두 개 (0과 1) 있습니다.
- 또한, 인자로 주어진 정수를 7로 나눈 나머지를 구하는 함수 (`` `rem` 7 ``) 혹은 (`flip rem 7`)의 고정점은 일곱 개 (0, 1, ..., 6) 이고,
- 항등 함수 `id`는 모든 인자들이 전부 다 고정점입니다.

당연히, 고정점에 대해서는 해당 함수를 몇 번 반복 적용해도 그 결과값이 변하지 않습니다. 다시 말해서, 만일 p가 f의 고정점이라면, $f\ p = p$일 뿐만 아니라 심지어 모든 자연수 n에 대해서 언제나 $f^n\ p = p$라는 뜻입니다.

1.1 고정점 도달

어떤 함수 f가 고정점이 하나 이상 있는 함수라고 할 때, 어떤 값 x가 f의 고정점이 아니더라도, f를 이 x에 계속 반복 적용하다보면 어느 순간 f의 고정점 중 하나에 "도달"하는 경우도 간혹 있습니다. 즉, 만일 f를 x에 반복 적용하는 각 단계마다 다음과 같은 상황이 벌어졌다면:

$$
\begin{array}{rlll}
\text{1회 적용 결과} & f^1\,x \neq f^0\,x & \text{즉,} & f\,x \neq x \\
\text{2회 적용 결과} & f^2\,x \neq f^1\,x & \text{즉,} & f\,(f\,x) \neq f\,x \\
\text{3회 적용 결과} & f^3\,x \neq f^2\,x & \text{즉,} & f\,(f^2\,x) \neq f^2\,x \\
\text{4회 적용 결과} & f^4\,x \neq f^3\,x & \text{즉,} & f\,(f^3\,x) \neq f^3\,x \\
 & \vdots & & \\
c-1\text{회 적용 결과} & f^{c-1}\,x \neq f^{c-2}\,x & \text{즉,} & f\,(f^{c-2}\,x) \neq f^{c-2}\,x \\
c\text{회 적용 결과} & f^{c}\,x \neq f^{c-1}\,x & \text{즉,} & f\,(f^{c-1}\,x) \neq f^{c-1}\,x \\
c+1\text{회 적용 결과} & f^{c+1}\,x = f^{c}\,x & \text{즉,} & f\,(f^{c}\,x) = f^{c}\,x \\
c+2\text{회 적용 결과} & f^{c+2}\,x = f^{c+1}\,x & \text{즉,} & f\,(f^{c+1}\,x) = f^{c+1}\,x \\
c+3\text{회 적용 결과} & f^{c+3}\,x = f^{c+2}\,x & \text{즉,} & f\,(f^{c+2}\,x) = f^{c+2}\,x \\
 & \vdots & &
\end{array}
$$

이는 $c+1$회 반복 적용을 통해 $f^c\,x$라는 고정점에 도달한 것입니다. (주의: 고정점은 반복 적용 횟수 $c+1$도 아니고 윗첨자 c는 더더욱 아니며, 이 단계의 인자(이자 동시에 리턴값)인 $f^c\,x$입니다.) 그리고 앞서 말했던 성질 때문에, 이렇게 한 번 고정점에 도달하면 그 이후로는 반복 적용을 계속해도 더이상 결과값이 변하지 않고 그대로 **고정**됩니다. 즉, 만일 $f^c\,x$가 f의 고정점이라면,

$$f^c\,x \;=\; f^{c+1}\,x \;=\; f^{c+2}\,x \;=\; f^{c+3}\,x \;=\; \cdots$$

입니다. (애초에 왜 "고정"점이라고 불리는지 이제 눈치챘을 겁니다.)

예를 들어, 인자로 주어진 정수를 7로 나눈 몫과 나머지 중 큰 쪽을 리턴하는 `Integral i => i -> i` 타입의 함수는 다음과 같이 여러가지 방식으로 만들 수 있는데:

- `(\x -> max (quot x 7) (rem x 7))`
- `(\x -> uncurry max (quotRem x 7))`
- `(\x -> uncurry max (flip quotRem 7 x))`
- `(\x -> (uncurry max . flip quotRem 7) x)`
- `(uncurry max . flip quotRem 7)`

이 함수를 초기 인자 42000에 20번 반복 적용해보면 아래와 같이 5회째 반복 적용하는 시점부터 결과가 계속 3으로 고정됩니다:

[*WinGHCi window, module* HW5 *loaded*]

```
*HW4> progress (uncurry max . flip quotRem 7) 20 42000
[42000,6000,857,122,17,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3]
it :: Integral t => [t]
*HW4>
```

즉, 42000으로부터 시작해서 도달 가능한 이 함수의 고정점은 3입니다.

함수 f의 고정점이 두 개 이상일 경우, 이 함수 f를 어떤 초기 인자에 반복 적용하느냐에 따라 (즉, 처음에 어떤 값으로부터 출발하느냐에 따라) 서로 다른 고정점에 도달하기도 합니다. 마침 위에서 예로 들었던 "7로 나눈 몫과 나머지 중 큰 쪽을 리턴하는 함수"가 바로 고정점이 두 개 이상(정확히는 0부터 6까지 일곱 개)인

함수였으며, 반복 적용을 시작하는 초기 인자가 42000이 아니라 49000일 경우 다음과 같이 또다른 고정점인 6에서 고정됩니다.

[*WinGHCi window, module* HW5 *loaded*]

```
*HW4> progress (uncurry max . flip quotRem 7) 20 49000
[49000,7000,1000,142,20,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6]
it :: Integral t => [t]
*HW4>
```

물론, 하나 이상의 고정점이 있는 어떤 함수를 어떤 초기 인자 x에 반복 적용하면 분명히 고정점 중 하나에 도달하는 반면, 이 함수를 또다른 어떤 초기 인자 y에 반복 적용하면 영원히 아무 고정점에도 도달하지 못하는 경우도 있습니다.

가장 간단한 예는 (2*) 함수입니다. 이 함수의 그래프는 $y = 2x$로서, $y = x$와 원점에서만 만나므로 고정점은 0 하나입니다. 이 함수를 초기 인자 0에 반복 적용하면 아예 처음부터 고정점에 도달해있는 상태지만:

[*WinGHCi window, module* HW5 *loaded*]

```
*HW4> progress (2*) 20 0
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
it :: Num t => [t]
*HW4>
```

초기 인자로 0 이외의 다른 수를 주면 양수는 큰 쪽으로, 음수는 작은 쪽으로 발산할 뿐이어서 영원히 고정점에 도달하지 못합니다.

[*WinGHCi window, module* HW5 *loaded*]

```
*HW4> progress (2*) 20 3
[3,6,12,24,48,96,192,384,768,1536,3072,6144,12288,24576,49152,98304,196608,393216,786432,
1572864,3145728]
it :: Num t => [t]
*HW4> progress (2*) 20 (-3)
[-3,-6,-12,-24,-48,-96,-192,-384,-768,-1536,-3072,-6144,-12288,-24576,-49152,-98304,-1966
08,-393216,-786432,-1572864,-3145728]
it :: Num t => [t]
*HW4>
```

살짝 다른 경우로, 발산하지는 않지만 수렴하거나 고정점에 다다르지도 못한 채 몇 개의 값들 사이를 빙빙 도는 사이클이 만들어지는 경우도 있습니다. 예를 들어, (다소 극단적인 예이긴 하지만) 인자로 주어진 정수가 짝수면 그 인자를 반으로 나누고 홀수면 그 인자에서 3을 빼는 `Integral i => i -> i` 타입의 함수

```
(\x -> if even x then div x 2 else x - 3)
```

는[2] 한 개의 고정점 (0)이 있습니다. 이 함수를 384에 반복 적용하면 고정점인 0에 도달하지만, 383에 반복

[2] 이 람다 함수 정의 코드 속에 등장하는 "`if`", "`then`", "`else`"는 함수 이름이나 변수 이름 같은 식별자(identifier)가 아니라 (마치 "`data`"나 "`class`", "`->`", "`=>`" 등처럼) Haskell의 특수한 문법 요소여서 juxtapositioning에 참여하지 않고 따로 처리됩니다. 즉, 위 코드는 자동적으로 다음과 같이 파싱됩니다:

```
(\x -> (if (even x) then (div x 2) else (x - 3)))
```

일반적으로, 이 문서의 Haskell 코드 중 굵고 진한 녹색 글씨로 표시된 것들은 통상적인 식별자나 연산자가 아니라 Haskell의 특수한 문법 요소들입니다.

적용하면 어느 시점부터 사이클 속에 진입해서 영원히 빙빙 돕니다:

[*WinGHCi window, module* HW5 *loaded*]

```
*HW4> progress (\x -> if even x then div x 2 else x - 3) 20 384
[384,192,96,48,24,12,6,3,0,0,0,0,0,0,0,0,0,0,0,0,0]
it :: Integral a => [a]
*HW4> progress (\x -> if even x then div x 2 else x - 3) 20 383
[383,380,190,95,92,46,23,20,10,5,2,1,-2,-1,-4,-2,-1,-4,-2,-1,-4]
it :: Integral a => [a]
*HW4>
```

이 숙제의 첫번째 문제(와 어쩌면 기말 시험 문제 중 일부)를 풀기 위한 필수 개념 설명은 이 위까지로 충분합니다. 이 지점부터 첫번째 문제 전까지의 내용은 기왕 고정점 개념과 고정점에 도달하는 개념을 설명한 김에 추가로 덧붙이는 참고 사항들이니, 시간이 부족하면 (아예 영원히 안 읽지는 말고) 나중에 천천히 읽어도 됩니다.

또 조금 다른 경우로, 고정점에 완전히 도달하지는 못하지만 어떤 특정 값에 (그 값이 고정점이든 아니든) 끝없이 수렴하는 경우도 있습니다. 가장 간단한 예들 중 하나는 주어진 **(수학적) 실수** 인자를 제곱하는 함수

$$\lambda x^{\mathbb{R}}.x^2 : \mathbb{R} \to \mathbb{R}$$

로서, 이 함수의 그래프 $y = x^2$은 그래프 $y = x$와 두 좌표($\langle 0, 0\rangle$과 $\langle 1, 1\rangle$)에서 만나므로 고정점은 두 개(0과 1)입니다. 또한, x 값의 개구간 $(0, 1)$ 범위에서 이 함수의 그래프 $y = x^2$은 항상 x축보다는 위에 있고 그래프 $y = x$보다는 아래에 있으므로, 이 함수를 0보다 크고 1보다 작은 실수 r에 적용하면 반드시 0보다 크고 "그 인자" r보다 작은 실수를 리턴합니다.[3]

$$\forall r^{\mathbb{R}}, 0 < r < 1 \to 0 < (\lambda x.x^2)\ r < r$$

위 식의 결론부는 개구간 $(0, 1)$ 사이의 모든 실수 r에 대해 성립하므로, 결론부의 r 자리에 (개구간 $(0, 1)$ 사이에 있음이 확인된) $(\lambda x.x^2)\ r$를 넣어도 성립하며:

$$\forall r^{\mathbb{R}}, 0 < r < 1 \to 0 < (\lambda x.x^2)\ ((\lambda x.x^2)\ r) < (\lambda x.x^2)\ r$$

이 식은 다음과 같이 정리될 수 있습니다:

$$\forall r^{\mathbb{R}}, 0 < r < 1 \to 0 < (\lambda x.x^2)^2\ r < (\lambda x.x^2)^1\ r$$

이 식의 r 자리에 다시 $(\lambda x.x^2)\ r$를 넣어가며 계속 재귀적으로 진행하면 다음과 같은 식이 도출됩니다:

$$\forall r^{\mathbb{R}}, 0 < r < 1 \to \forall n^{\mathbb{N}}, 0 < (\lambda x.x^2)^{n+1}\ r < (\lambda x.x^2)^n\ r$$

즉, 0보다 크고 1보다 작은 실수를 초기 인자로 삼아서 이 함수를 계속 반복 적용하면, 그 결과는 점점 작아지

[3] 편의상, 이 문서에서는 "$(A < B) \wedge (B < C)$"를 "$A < B < C$"라고 축약해서 표기하겠습니다.

면서 0에 수렴하지만 결코 정확히[4] 0이 되지는 않습니다.[5]

$$\forall r^{\mathbb{R}}, 0 < r < 1 \rightarrow \left(\lim_{n\to\infty} (\lambda x.x^2)^n\ r = 0\right) \wedge \left(\forall n^{\mathbb{N}}, (\lambda x.x^2)^n\ r \neq 0\right)$$

참고로, 이렇게 무한 개념을 적용하여 함수의 (정밀한 수렴 근사값이 아닌) 정확한 수렴값을 논리적으로 도출해내는 일은 이번 학기에는 다루지 않겠습니다. (실은, 함수를 무한 반복 적용할 때의 정확한 수렴값에 관한 개념은 이번 학기에 다루지 않기로 한 "람다 텀만으로 재귀 함수 만들기"와 살짝 연결된 개념입니다.) 이번 학기에는 "수렴하는 경우도 있다"라고 개념만 이해해 두면 됩니다.

또한 참고로, Haskell에서 부동소수점 수의 부동소수점 수 제곱을 계산해주는

```
(**) :: Floating a => a -> a -> a
```

연산자를 이용해서 위의 "(수학적) 실수"에 대한 제곱 함수를 비슷하게 흉내내서 계산해보면 다음과 같이 나오는데:

[*WinGHCi window, module* HW5 *loaded*]

```
*HW4> progress (**2) 25 0.999
[0.999,0.998001,0.996005996001,0.9920279440699441,0.9841194418156402,0.9684910757595271,0
.937974963825846,0.8797970327640972,0.7740428188605099,0.5991422854295241,0.3589714781897
133,0.1288605221537078 3,1.6605034169726227e-2,2.7572715977777556e-4,7.6025466639118 97e-8,
5.779871577695792e-15,3.3406915454655646e-29,1.1160220001945103e-57,1.2455051049181556e-1
14,1.551282966377186e-228,0.0,0.0,0.0,0.0,0.0,0.0,0.0]
it :: Floating t => [t]
*HW4>
```

결과에 보이듯이

- 유효숫자의 갯수(즉, 정밀도)의 최대 한도가 일정하게 (10진법 기준 약 16–17자리로) 고정되어 있어서 그 이상으로 늘어나지 못하고,
- 지수표기의 지수값(즉, 스케일 범위)도 최대/최소가 일정하게 제한되어 있어서 이 최소값을 벗어나면 (정확한 0과는 다른, 근본적으로 근사값인) "0.0"이라는 부동소수점 수에 도달하여 반복되며,
- 심지어, 위 결과에는 잘 드러나지 않지만 10진법 유한소수와 2진법 순환소수 사이를 오가는 중에도 오차가 발생합니다.

이는 IEEE 754 부동소수점 수의 정밀도 (유효숫자 갯수) 및 스케일 (지수 범위) 한계 때문이며, Haskell 뿐만 아니라 어떤 언어에서라도 하드웨어(CPU)에 내장된 고효율 부동소수점 연산 유닛을 직접 이용하도록 설계된 (주로 내장) 자료형들[6]은 이 한계에서 자유로울 수 없습니다. 이 한계를 돌파하여 '원하는 만큼 정밀'하거나 '원하는 만큼 크거나 작은' 근사값을 제대로 계산하려면, 그렇게 동작하는 자료형과 연산을 직접 프로그래밍해서

[4] "정확(exact)"은 yes/no 개념이고, "정밀(precise)"은 정도 개념입니다.

[5] 무한대(∞)는 (논란의 여지가 있을지도 모르지만, 적어도 구성주의적 관점에서는 확실히) 수(數)가 아닙니다. 즉, "모든 자연수 n에 대해서"라고 말할 때 무한대는 그 '모든 자연수'에 포함되지 않습니다.

[6] C/Java의 단정도 `float`과 배정도 `double`, Python의 배정도 `float`, Haskell의 단정도 `Float`과 배정도 `Double` 등.

쓰거나, 각 언어마다 소프트웨어적으로 만들어서 (주로 표준 라이브러리나 외부 라이브러리로) 추가 제공하는 고정밀도/임의정밀도 부동소수점 수 자료형/연산[7]을 사용해야 합니다.

다만, 실질적으로는 유한 집합까지, 개념적으로도 가산 무한 집합[8]까지만 다룰 수 있는 컴퓨터와 소프트웨어 상에는 불가산 무한인 수학적 실수가 온전히는 존재할 수 없는데, 그럼에도 불구하고 이 (**2) 함수를 이용해서 앞서 "수학적 실수" 위에서 정의했던 함수 $\lambda x^{\mathbb{R}}.x : \mathbb{R} \to \mathbb{R}$를 억지로 흉내내 놓고 수학적 실수의 성질을 그대로 담아내지 못하는 측면을 지적하기에는 기계 입장에서 억울할 수 있습니다.

따라서, 아예 처음부터 (**2)를 그냥 부동소수점 수 위에 정의된 함수인 것으로 간주하면 (즉, 정의역과 공변역을 부동소수점 수로 간주하고 부동소수점 수의 특성만 생각하면) 부동소수점 수 0.0은 이 함수의 올바른 고정점이 맞습니다.

드디어 문제입니다: 4차 숙제에서 만들었던 `iterApp`처럼 주어진 초기 인자에 주어진 단항 함수를 반복 적용하되, 반복 횟수까지 인자로 받아서 그 횟수 만큼 반복 적용했던 삼항 함수 `iterApp`과는 달리, 반복 횟수를 인자로 받지 않는 대신 고정점에 도달할 때까지 몇 번이든 알아서 계속 반복 적용하다가

- 고정점에 도달하면 멈추고 그 고정점을 리턴하고
- 고정점에 도달하지 못하면 계속 무한 재귀에 빠지는

이항 함수

```
fixpoint :: Eq t => (t -> t) -> t -> t
```

를 정의하십시오. 단, **반드시 꼬리 재귀 형태로 정의**해야 합니다.

무엇을 정의하라는 것인지 이해를 돕기 위해 약간 더 부연하자면, 함수 `fixpoint`를 두 인자에 적용한 "`fixpoint f x`"의[9] 계산 결과는 결국 "초기 인자 `x`에 `f`를 반복 적용했을 때 (만일 있고, `x`로부터 도달 가능하다면) 도달하게 되는 `f`의 고정점"입니다. 예를 들어, 올바르게 정의된 `fixpoint`는 다음과 같이 동작해야 합니다:

[*WinGHCi window, module* HW5 *loaded*]

```
*HW5> fixpoint (uncurry max . flip quotRem 7) 42000
3
it :: Integral t => t
*HW5> fixpoint (uncurry max . flip quotRem 7) 49000
6
it :: Integral t => t
*HW5>
```

물론, 이 `fixpoint` 함수는 올바른 타입의 인자들이 주어지더라도 그 계산이 종료된다는 보장이 없는 위험한 함수입니다. 만일

---

[7] C/C++ 용 외부 라이브러리인 GMP, Java 표준 라이브러리의 BigDecimal, Python 표준 라이브러리의 decimal, Haskell 외부 라이브러리인 numbers 등.

[8] '모든 자연수들의 집합'인 $\mathbb{N}$과 크기가 같은 무한 집합. 달리 말하자면, 해당 집합의 모든 원소들과 $\mathbb{N}$의 모든 원소들 사이에 1:1 대응을 만들어줄 방법(알고리즘)이 존재하는 집합. 또 달리 말하자면, 해당 집합의 원소들 중 어떤 원소를 지목해도 그 원소가 해당 집합 속에서 "몇 번째" 원소인지를 반드시 (서로 겹치지 않게 배타적으로) 정해서 말할 수 있는 집합. (예: 모든 짝수들의 집합, 모든 정수들의 집합, 모든 유리수들의 집합, 모든 '정수의 정수 제곱근'들의 집합, 모든 '실수부와 허수부가 둘 다 유리수인 복소수'들의 집합 등)

[9] "`fixpoint` of `f` from `x`"라고 읽습니다.

- 첫번째 인자 f가 고정점이 없는 함수거나
- f가 하나 이상의 고정점이 있는 함수지만 두번째 인자 x로부터 반복 적용을 시작해서는 어떤 고정점에도 도달할 수 없을 경우

종료되지 못하고 무한 재귀에 빠집니다:

[*WinGHCi window, module* HW5 *loaded*]

```
*HW5> fixpoint (+1) 1
```

(이렇게 무한 재귀에 빠지도록 작성하면 안된다는 뜻이 아니라, 이런 경우에는 이렇게 무한 재귀에 빠지는 게 정상이니 이 현상을 신경쓰지 말고 작성하라는 뜻입니다.)

힌트: 이 fixpoint 함수의 타입이 왜 "(t -> t) -> t - t"가 아닌 "Eq t => (t -> t) -> t - t"일지 (즉, 어째서 아무 t 타입이어서는 안되고 반드시 Eq 클래스에 소속된—동치 비교가 가능한—타입이어야만 하는지) 생각해보기 바랍니다. 이 함수는 where 절로 (인자의 갯수가 다른) 별도의 함수를 정의할 필요 없이 (마치 iterApp이 그랬던 것처럼) 이 함수 자체를 꼬리 재귀 형태로 작성할 수 있습니다.

1.2 고정점 도달열

4차 숙제에서 만들었었던 iterApp이 함수를 n번 반복 적용한 최종 결과만 달랑 리턴하는 반면, progress는 0번 반복 적용한 결과부터 n번 반복 적용한 결과까지 총 $n+1$개의 원소가 들어있는 리스트를 리턴했었습니다. 즉, progress는 iterApp의 "수열"로서의 짝이었습니다. 비슷하게, fixpoint에게도 "수열"로서의 짝인 fixProgress를 만들어줄 수 있습니다. fixpoint가 고정점에 도달할 때까지 조용히 반복 적용을 계속하다가 (고정점이 있고, 도달 가능하다면) 최종적으로 도달한 고정점만 리턴하는 반면, fixProgress는 0번 반복 적용한 결과부터 고정점까지의 각 적용 단계의 값들이 들어있는 (길이를 미리 특정할 수 없는) 리스트를 리턴한다는 뜻입니다.

이 fixProgress가 어떻게 동작해야 하는지에 대해 보다 상세하게 규칙을 정하겠습니다.

- 먼저, fixProgress는 (progress와 마찬가지로) 어떤 경우에도 결코 빈 리스트를 리턴하지 말아야 합니다. "fixProgress f x"라는 리스트의 첫번째 원소는 언제나 f^0 x, 즉, 초기 인자인 x 자신이며, 따라서 "fixProgress f x"라는 리스트의 길이는 언제나 1 혹은 그 이상이어야 합니다.
- 둘째, "fixProgress f x"라는 리스트는 유한 리스트일 수도 있지만 무한 리스트일 수도 있습니다. 정확히는, 함수 f가 고정점이 없는 함수거나, 초기 인자 x로부터 반복 적용을 시작해서는 f의 고정점(들 중 하나)에 도달할 수 없는 경우에 무한 리스트가 됩니다.
- 셋째, 만일 "fixProgress f x"가 유한 리스트라면, 이 리스트의 마지막 원소는 반드시 함수 f의 (x로부터 도달 가능한) 고정점이어야 합니다. 즉, 이 소절 첫 단락에서 "고정점까지의 각 적용 단계의 값들이 들어있다"고 했던 말 속의 '고정점까지'에는 고정점 자체가 확실하게 포함됩니다.
- 넷째, "fixProgress f x"라는 (유한 혹은 무한) 리스트의 어떤 두 인접한 원소도 서로 같지 않아야[10] 합니다. 고정점에 도달하기 전까지는 (고정점의 정의 상) 어떤 두 인접한 원소도 서로 같지 않을 수 밖에

[10] "같지 않아야"한다고 했지 "달라야"한다고 하지 않았다는 점도 (이번 숙제나 대부분의 상황에서는 사실은 신경 안 써도 되지만) 참고 삼아 기억해두기 바랍니다. Haskell 내장 (built-in) 타입들 중 Eq 클래스에 소속된 대부분의 타입들의 값들은

$$((\texttt{not}\ (\texttt{x}\ ==\ \texttt{y})) \equiv (\texttt{x}\ /=\ \texttt{y})) \wedge ((\texttt{not}\ (\texttt{x}\ /=\ \texttt{y})) \equiv (\texttt{x}\ ==\ \texttt{y}))$$

없고, 고정점에 도달한 시점부터는 그 고정점이 (마지막 원소로) 딱 한 번만 리스트에 들어있어야 한다는 뜻입니다.

이 규칙들에 따르면, 초기 인자 자신이 고정점인 경우에는 정확히 길이 1 짜리 리스트가 만들어지며:

[*WinGHCi window, module* HW5 *loaded*]

```
*HW5> fixProgress (*2) 0
[0]
it :: (Eq t, Num t) => [t]
*HW5>
```

5번째 반복 적용($f^5\ x$)에 의해 고정점에 도달했다면 길이 6 짜리 리스트가 만들어져야 합니다:

[*WinGHCi window, module* HW5 *loaded*]

```
*HW5> progress (uncurry max . flip quotRem 7) 20 42000
[42000,6000,857,122,17,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3]
it :: Integral t => [t]
*HW5> fixProgress (uncurry max . flip quotRem 7) 42000
[42000,6000,857,122,17,3]
it :: Integral t => [t]
*HW5>
```

문제입니다: 이 소절에서 정한 규칙대로 동작하는 이항 함수

```
fixProgress :: Eq t => (t -> t) -> t -> [t]
```

를 정의하십시오. 단, 내장 함수 `iterate`나 위에서 정의했던 `fixpoint`, 숙제 코드에 주어진 `iterApp`, `progress` 중 어떤 것도 사용하지 말고 정의하십시오. 또한, 이 `fixProgress` 함수는 굳이 꼬리 재귀로 작성할 필요 없으며, 오히려 그러지 않기를 권장합니다. (이유는 수업 때 설명하겠습니다.)

2 거품 정렬

2.1 거품기

리스트 속의 원소들에 대한 거품 정렬(bubble sorting)을 이야기하기 전에, 이 거품 정렬의 부분 문제로서 "거품기"부터 먼저 생각해보겠습니다.

단항 함수 `bubble`은 동치 비교가 가능한 값들이 들어있는 리스트 하나를 인자로 받아서, 만일 리스트가 비어있거나 원소가 하나 뿐이면 그 리스트를 그대로 리턴하고, 원소가 둘 이상이면 서로 인접한 두 원소를 앞에서부터 차례대로 비교하면서 크기 순서대로 놓여있으면 그대로 두고 순서가 반대면 둘의 위치를 바꾸는 연산입니다. 주의할 점은:

가 성립하도록 정의되어 있긴 하지만, 어디까지나 "대부분"일 뿐 이것이 항상 보장되지는 않기 때문입니다. 너무 지협적인 주의사항이라고 느낄지도 모르고 막상 이번 숙제의 답을 작성하는 데에는 실질적으로 상관이 없지만, 간혹 논리적인 비약으로 인해 자칫 실수할 수 있는 부분이어서 주석으로 언급해둡니다.

- 비교할 대상은 인접한 두 원소지만, 다음번 비교를 위해 비교할 위치를 이동할 때에는 두 자리씩 건너뛰는 것이 아니라 한 자리만 옮겨가야하며,
- 인접한 두 원소의 자리를 바꿨을 경우, 다음 자리로 옮겨가서 비교할 때에는 조금 전에 자리를 바꾼 결과가 반영된 상태로 비교해야합니다.

예를 들어, 리스트 `[3,6,1,5,4,7,8,6]`을 `bubble`하는 세부 과정은 다음과 같습니다:

- `[3,6,1,5,4,7,8,6]` 시작: 주어진 리스트의 맨 앞 3과 6 비교
- `[3,6,1,5,4,7,8,6]` 3≤6이므로 그대로 두고 다음 위치인 6과 1 비교
- `[3,1,6,5,4,7,8,6]` 6>1이므로 순서를 바꾸고 다음 위치인 6과 5 비교
- `[3,1,5,6,4,7,8,6]` 6>5이므로 순서를 바꾸고 다음 위치인 6과 4 비교
- `[3,1,5,4,6,7,8,6]` 6>4이므로 순서를 바꾸고 다음 위치인 6과 7 비교
- `[3,1,5,4,6,7,8,6]` 6≤7이므로 그대로 두고 다음 위치인 7과 8 비교
- `[3,1,5,4,6,7,8,6]` 7≤8이므로 그대로 두고 다음 위치인 8과 6 비교
- `[3,1,5,4,6,7,6,8]` 8>6이므로 순서를 바꾸고 종료

즉, 리스트 `[3,6,1,5,4,7,8,6]`을 `bubble`한 결과는 즉, 리스트 `[3,1,5,4,6,7,6,8]`입니다.

[WinGHCi window, module HW5 loaded]

```
*HW5> bubble [3,6,1,5,4,7,8,6]
[3,1,5,4,6,7,6,8]
it :: (Ord a, Num a) => [a]
*HW5>
```

문제입니다: 위에서 제시한 단항 함수

```
bubble :: Ord a => [a] -> [a]
```

을 작성하십시오. 혹시 정 필요하다면 리스트의 첫번째 원소를 리턴하는 함수 `head`나 리스트의 첫번째 원소를 제외한 나머지 뒷부분 리스트를 리턴하는 함수 `tail`을 사용해도 틀리진 않지만, 가능한 한 이들을 사용하지 않기를 권장합니다. 또한, 이 함수는 굳이 꼬리 재귀로 작성할 필요 없습니다. 참고로, Haskell에서 순서(대소) 관계를 비교하는 중위 연산자는 `<=`, `<`, `>=`, `>`이며, 이들의 타입은 공통적으로 `Ord a => a -> a -> Bool` 입니다.

2.1.1 힌트: 중첩된 패턴

숙제 코드 파일에 정의되어있는 `delSameAdj` 함수는 (동치 비교가 가능한 타입의 값들의) 리스트 하나를 인자로 받아서, 인자 리스트 속의 **인접한** 둘 이상의 원소들이 서로 같을 때마다 그 "인접한 같은 원소들" 중 첫번째 (가장 왼쪽) 원소 하나만 남기고[11] 나머지는 모두 지운 리스트를 리턴합니다. 예를 들어, 다음과 같이 동작합니다:

[11] 같은 원소들이라고 했으니 그 중 어느 원소를 남기고 어느 원소를 없애든 마찬가지 아닌가 싶을지도 모르지만, 실제로는 그렇지 않을 수도 있기 때문에 이렇게 딱 짚어서 말한 것입니다. 예를 들어, 만일 유리수 타입이 간단히 "양수 분의 정수"라고만 정의되어

[WinGHCi window, module HW5 loaded]

```
*HW5> delSameAdj "Haskell Brooks Curry"
"Haskel Broks Cury"
it :: [Char]
*HW5>
```

이 함수를 언급하는 이유는 이번 숙제를 풀 때 이 함수를 사용하라는 뜻이 아니라, 이 함수를 정의할 때 사용된 몇 가지 테크닉들을 참고하라는 뜻입니다. 숙제 코드 속에 다음과 같이[12] 정의되어 있는데:

file: HW5.hs

```
41 delSameAdj :: Eq a => [a] -> [a]
42 delSameAdj (x:x':xs)
43     | x == x'    =      delSameAdj (x :xs)
44     | otherwise  =  x : delSameAdj (x':xs)
45 delSameAdj xs     =  xs
46 -- delSameAdj [x]   =  [x]
47 -- delSameAdj []    =  []
```

위 정의 중 42행에서 사용된 "(x:x':xs)"라는 패턴에 주목하기 바랍니다.

이미 지난 숙제에서도 여러 번 사용해봤으니 이제 상당히 익숙하겠지만, 리스트를 인자로 받는 함수를 정의할 때 리스트 인자 자리에:

- 단순히 매개변수 이름만 (예를 들어 "xs"라고) 쓰면 모든 리스트가 그 자리에 대응되면서 그 리스트에 (그 정의문 내부에서만 임시로) xs라는 이름이 붙고
- "[]"라는 패턴을 사용하면 빈 리스트만 이 자리에 대응되고
- "[x]"나 "(x:[])"라는 패턴을 사용하면 원소가 정확히 한 개인 리스트만 그 자리에 대응되면서 그 유일한 원소에 (역시 해당 정의문 내부에서만 임시로) x라는 이름이 붙으며
- "(x:xs)"라는 패턴을 사용하면 원소가 최소 한 개 이상인 리스트만 그 자리에 대응되면서 그 리스트의 첫번째 원소에 x, 나머지 뒷부분 리스트에 xs라는 이름이 임시로 붙습니다.

그런데, 온라인 입문서 4장 초반에 나오고 수업 시간에 설명했듯이, 패턴은 임의의 깊이로 **중첩**될 수도 있습니다. 예를 들어, 바로 위 목록 중 마지막 항목에서 설명했던 패턴 "(x:xs)"의 나머지 뒷부분 리스트 xs 자리에 (아무 리스트나 다 대응시키는 대신) 또다시 최소 하나 이상의 원소를 갖고있는 리스트만 대응시키려면, 이 자리에 또다시 그런 패턴을 쓸 수 있다는 뜻입니다. 즉, "(x:(x':xs))"라는 패턴이나 혹은 (이항 중위 자료생성자 ":"는 원래 오른쪽부터 먼저 묶이므로, 불필요한 괄호 없이) "(x:x':xs)"라는 패턴은 최소 둘 이상의 원소를 갖고있는 리스트에만 성공적으로 대응되고, 이때 첫번째 원소에 x, 두번째 원소에 x', 나머지 리스트에 xs라는 이름을 (해당 정의문 내부에서만 임시로) 붙입니다.

있어서 약분되지 않은 유리수(예: $6/9$)도 올바른 유리수라고 약속되어있고, 이 유리수 타입을 Eq 클래스에 소속시키면서 유리수에 대한 == 연산을 흔히 사용하는 유리수에 대한 산술적 동치 관계처럼 "두 유리수의 분자, 분모를 대각선끼리 곱한 것이 서로 같으면 두 유리수는 같다"라고 정의했다면, 리스트 속의 인접한 세 원소 $2/3, 4/6, 6/9$ 중 무엇을 없애고 무엇을 남기냐에 따라 이후 추가 연산의 결과(예: 리스트 각 원소의 분자만 취한 결과로 그 원소 자리가 2가 될지 4가 될지 6이 될지)가 달라질 수도 있습니다.

[12] 단, 행 번호는 이 문서 상에서의 설명의 편의를 위해 임의로 붙였을 뿐이며, 실제 코드의 행 번호와 다를 수도 있습니다.

2.1.2 힌트: 원소 순서 바꾸기 vs. 원소 순서가 다른 새 리스트 만들기

앞 소절에서 언급했던 `delSameAdj` 함수의 정의에서 또 한 가지 참고할 점은, 만들고자 하는 함수가 비록 개념적으로는 인자로 받은 리스트 속의 몇몇 원소들을 제거하는 함수긴 하지만, 그 함수를 실제로 정의하는 과정에서는 인자로 받은 리스트 원본을 결코 수정(원소 삭제)하지 않고 그저 **삭제 대상 원소들만 안 들어있고 나머지 원소들은 똑같이 들어있는 새로운 리스트**를 새로 생성할 뿐이라는 점입니다. 함수형 언어에는 "상태를 바꾼다"는 개념이 없기 때문이며, 함수형 언어로 문제를 풀 때에는 그 문제를 **어떻게** 풀지(즉, 어떤 절차에 따라 상태를 계속 바꿔갈지)를 말하는 것이 아니라 그 문제의 답이 **무엇인지**를 말해야 합니다.

이 소절에서 참고삼아 살펴보고 있는 `delSameAdj` 함수의 정의도, 인자로 주어진 리스트에서 인접한 같은 값들을 하나만 남기고 제거하려면 어떤 과정을 거치며 어떻게 해야하는지를 말하고 있는 정의가 아니라, 인자로 주어진 리스트에서 인접한 같은 값들을 하나만 남기고 제거한 **최종 결과 리스트란 과연 무슨 리스트인지**를 말하고 있는 정의입니다. (구체적으로, "원소가 최소 둘 이상 들어있는 리스트의 첫번째 원소를 `x`라고 하고 두번째 원소를 `x'`이라고 하고 이후 (나머지) 리스트를 `xs`라고 할 때, 이 (전체) 리스트를 `delSameAdj`한 결과 리스트란, 만일 `x == x'`이면 `(x:xs)`라는 리스트를 `delSameAdj`한 결과 리스트고, 그렇지 않으면 첫번째 원소가 `x`고 이후 리스트는 `(x':xs)`라는 리스트를 `delSameAdj`한 결과 리스트인 리스트다"라고 말하고 있습니다.)

지금 숙제로 만들어야 하는 `bubble` 함수가 비록 개념적으로는 몇몇 인접한 두 원소들의 위치를 서로 뒤바꾸는 함수긴 하지만, 이 함수를 정의할 때에는 인자 리스트 원본을 "수정"해서 원소들의 위치를 실제로 바꾸는 것이 아니라, "인자 리스트와는 몇몇 원소들의 순서가 다른 새 리스트로 계산되는 계산식"을 (즉, 그런 새 리스트가 무슨 리스트인지를) 말해야합니다. 즉, `delSameAdj` 함수의 정의 방식이 크게 참고가 될 것입니다.

2.1.3 기타 팁

힌트라고 하기에는 아주 작은 팁에 가까운데, 앞서 보여준 `delSameAdj` 함수의 정의 중 42행보다 45행이 늦게 확인되기때문에, 45행에 사용된 단순한 매개변수 `xs`는 결국 "원소가 최소한 둘 이상인 리스트(42행에 대응된 리스트)가 아닌 나머지 모든 경우", 즉, 원소가 하나 이하인 리스트만을 뜻하게 됩니다.

실제로 위 정의에서 45행을 지우고 46–47행의 주석을 풀어도 똑같이 동작합니다. 다만, 이렇게 패턴 매칭의 횟수를 줄일 수 있는 경우도 있다는 것을 보이기 위해서 일부러 45행의 정의를 택했을 뿐입니다.

2.2 거품 정렬

거품 정렬(bubble sorting)이란 리스트 정렬 방식 중 하나로서, 주어진 리스트를 더이상 변화가 없을 때까지 반복적으로 `bubble`하여 정렬하는 방식입니다.

"연계" 문제입니다: 주어진 리스트를 거품 정렬하는 함수

```
bbSort :: Ord a => [a] -> [a]
```

를 **반드시 한 줄로, 가능한 한 짧게** 정의하십시오. 단, "`if ... then ... else ...`" 구문은 사용해선 안됩니다. 만일 필요하다면, 이 숙제에서 이미 본인이 다른 문제의 답으로서 작성했던 함수들(`bubble` 등)을 사용해도 됩니다. 또한, 정의문의 길이를 줄이는 데에 도움이 된다면 η-환원[13]을 활용해도 됩니다. 올바르게 정의된

[13] 정의되고있는 함수 바로 옆에 매개변수나 패턴을 위치시키는 "`foo x = bar baz x`" 라는 정의문은 실제로는 "`foo = \x -> bar baz x`" 라는 정의문의 syntactic sugar이므로, 이 정의문의 우변을 η-환원하면 "`foo = bar baz`" 가 됩니다.

`bbSort` 함수는 다음과 같이 동작해야 합니다:

[*WinGHCi window, module* HW5 *loaded*]

```
*HW5> bbSort []
[]
it :: Ord a => [a]
*HW5> bbSort [5]
[5]
it :: (Ord a, Num a) => [a]
*HW5> bbSort [6,5,8,7,4,0,5,2,6,5,9,8,1,7]
[0,1,2,4,5,5,5,6,6,7,7,8,8,9]
it :: (Ord a, Num a) => [a]
*HW5>
```

힌트: 왜 위와 같은 ("한 줄로" 등의) 제약 조건과 다른 문제의 답을 사용해도 된다는 안내가 붙어있을지, 문제의 의도를 잘 파악하기 바랍니다. 이 문서의 1절과 2절은 서로 관련 없거나 뜬금 없는 내용이 아닙니다. 이 문제가 "연계" 문제인 이유는 2.1절과 연계되는 점 외에도 한 가지 이유가 더 있습니다.

또한 참고로, 동치 비교가 가능한 원소들이 들어있는 리스트들끼리는 동치 비교가 가능하며, 규칙은 "두 리스트의 길이가 같고, 각 리스트의 같은 인덱스의 원소끼리 모두 서로 같으면, 두 리스트는 같다"입니다. (그래서 두 무한 리스트는 동치 비교가 가능하며 다르면—같은 인덱스의 원소끼리 서로 다른 경우가 발견되는 순간—확실히 다르다고 말해주지만, 같으면 같다고 말하지 못하고 무한 재귀에 빠집니다.) 그리고 `Ord` 클래스는 `Eq` 클래스를 함의하기 때문에 (`Ord` 클래스 소속 타입이라면 예외 없이 `Eq` 클래스에도 소속되어 있기 때문에) 결국 정렬 가능한 리스트들끼리는 서로 동치 비교도 항상 가능합니다.

2.2.1 추가 고려 사항

숙제 문제를 스스로 고민하고 풀어보는 데에 방해가 될까봐 (너무 심한 힌트가 될까봐) 이 문서에서는 간단히 언급만 할 수 밖에 없지만, 추후 수업 시간에 설명할 두 가지 추가 고려 사항이 있습니다.

첫째, 이 거품 정렬이 "언제나 반드시 종료하는지", "어째서 그런지"에 대해서 이 숙제의 풀이 시간 전까지 스스로 생각해보기 바랍니다. 계산의 종료 여부는 계산 이론 분야의 매우 중요한 화두 중 하나로서, 실제로 프로그램을 작성할 때에도 조심스럽게 계획하고 고려하며 설계/구현해야하는 부분이며, 필요하다면 충실한 테스트나 심지어 정형적인 검증(증명)까지 필요한 경우도 있습니다. 또한, 이는 추후 수업 시간에 설명할 "구조적 귀납(재귀) vs. 비구조적 귀납(재귀)"의 화두와도 직결됩니다.

둘째, 만일 본인이 성공적으로 "η-환원까지 적용된" 정의를 작성했고 제대로 잘 동작한다는 것을 확인했다면, 소스 코드 중 `bbSort`의 타입 선언 부분, 즉,

```
bbSort :: Ord a => [a] -> [a]
```

이라는 행을 지우거나 주석 처리한 후, Haskell이 `bbSort`의 타입을 자동적으로 추론해주기를 기대하며 WinGHCi 창에 다시 로드해보기 바랍니다. 어떤 현상을 목격할 것입니다. 또한, 그 상태에서 (`bbSort`의 타입 선언부가 여전히 주석 처리된 상태에서) 이번에는 η-환원만 하지 말고 (`bbSort`의 정의문을 η-환원되지 않은 형태로 되돌리고) 다시 한 번 로딩을 시도해보기 바랍니다. 재미있는 현상을 목격할 것이며, 이것이 학기 초에 "η-환원은 간혹 문제를 일으키는 경우가 있다"고 언급했던 부분의 실제 사례입니다.

2.3 거품 정렬 과정

거품 정렬 과정의 각 단계, 즉 `bubble`을 한 번 적용할 때마다의 중간 결과들을 모두 나열해보고 싶습니다.

역시 "연계" 문제입니다: 초기 인자 리스트로부터 거품 정렬 최종 결과까지의 전 과정을 (초기 인자와 최종 결과를 각각 처음과 끝에 딱 한 번 씩 포함해서) 리스트로 나열하는 단항 함수

```
bbSortProg :: Ord a => [a] -> [[a]]
```

를 이번에도 **반드시 한 줄로, 가능한 한 짧게** 정의하십시오. 역시 "`if ... then ... else ...`" 구문은 사용해선 안되며, 만일 필요하다면 이 숙제에서 이미 본인이 다른 문제의 답으로서 작성했던 함수들(`bubble` 등)을 사용해도 됩니다. 또한, 정의문의 길이를 줄이는 데에 도움이 된다면 η-환원을 활용해도 됩니다. 올바르게 정의된 `bbSortProg` 함수는 다음과 같이 동작해야 합니다:

[*WinGHCi window, module* HW5 *loaded*]

```
*HW5> bbSortProg [6,5,8,7,4,0,5,2,6,5,9,8,1,7]
[[6,5,8,7,4,0,5,2,6,5,9,8,1,7],[5,6,7,4,0,5,2,6,5,8,8,1,7,9],[5,6,4,0,5,2,6,5,7,8,1,7,8,9
],[5,4,0,5,2,6,5,6,7,1,7,8,8,9],[4,0,5,2,5,5,6,6,1,7,7,8,8,9],[0,4,2,5,5,5,6,1,6,7,7,8,8,
9],[0,2,4,5,5,5,1,6,6,7,7,8,8,9],[0,2,4,5,5,1,5,6,6,7,7,8,8,9],[0,2,4,5,1,5,5,6,6,7,7,8,8
,9],[0,2,4,1,5,5,5,6,6,7,7,8,8,9],[0,2,1,4,5,5,5,6,6,7,7,8,8,9],[0,1,2,4,5,5,5,6,6,7,7,8,
8,9]]
it :: (Ord a, Num a) => [[a]]
*HW5>
```

리스트들의 리스트를 만드는 함수이므로 이 결과가 맞긴 하지만, 알아보기기 힘듭니다. 따라서, `bbSortProg` 함수를 정의한 후 테스트해볼 때에는 다음과 같이 해보기 바랍니다 ("`mapM_`"의 밑줄은 오타가 아닙니다. 그대로 따라하셔요):

[*WinGHCi window, module* HW5 *loaded*]

```
*HW5> mapM_ print $ bbSortProg [6,5,8,7,4,0,5,2,6,5,9,8,1,7]
[6,5,8,7,4,0,5,2,6,5,9,8,1,7]
[5,6,7,4,0,5,2,6,5,8,8,1,7,9]
[5,6,4,0,5,2,6,5,7,8,1,7,8,9]
[5,4,0,5,2,6,5,6,7,1,7,8,8,9]
[4,0,5,2,5,5,6,6,1,7,7,8,8,9]
[0,4,2,5,5,5,6,1,6,7,7,8,8,9]
[0,2,4,5,5,5,1,6,6,7,7,8,8,9]
[0,2,4,5,5,1,5,6,6,7,7,8,8,9]
[0,2,4,5,1,5,5,6,6,7,7,8,8,9]
[0,2,4,1,5,5,5,6,6,7,7,8,8,9]
[0,2,1,4,5,5,5,6,6,7,7,8,8,9]
[0,1,2,4,5,5,5,6,6,7,7,8,8,9]
it :: ()
*HW5>
```

참고로, 여기서 중위 연산자 "`$`"는 아주 심플하게 왼쪽 피연산자를 오른쪽 피연산자에 적용해주는 연산자로서, `(f $ x)`는 언제나 `(f x)`와 같습니다. 예를 들어, 위 WinGHCi 입력을 이 연산자 없이 하려면 다음과 같이 입력하면 됩니다:

```
mapM_ print (bbSortProg [6,5,8,7,4,0,5,2,6,5,9,8,1,7])
```

이런 황당한 중위 연산자가 왜 필요한지 의아하겠지만, 이 연산자 덕분에 (특히, 한 행에서 주로 오른쪽에 포진한) 괄호들을 줄일 수 있는 경우가 많고, 여러 개의 `$`들을 이용해서 여러 쌍의 괄호를 줄이면 자칫 괄호를 잘못 짝짓는 실수도 자연스럽게 줄어듭니다.

`mapM_`의 타입은 다음과 같은데:

```
mapM_ :: (Foldable t, Monad m) => (a -> m b) -> t a -> m ()
```

이 수업의 목표 범위를 한참 벗어난 물건이고, 시험에도 당연히 등장하지 않거나 등장하더라도 내막을 몰라도 상관 없으니 자세히는 설명하지 않겠습니다. 그저, "`mapM_ print` *list*"라고 하면 *list*의 각 원소를 한 줄에 하나씩 `print` 해준다는 것만 기억하고 (숙제 결과 테스트 등에) 필요할 때 꺼내서 쓰면 됩니다.

문제는 몇 문제 안 되는데 설명이 길어서 힘들죠? 근데 기말고사 시험지도 이래요. =3=33