LAB 09

Mutable Trees, Efficiency

Mingxiao Wei
mingxiaowei@berkeley.edu

Mar 21, 2023

LOGISTICS 🏡

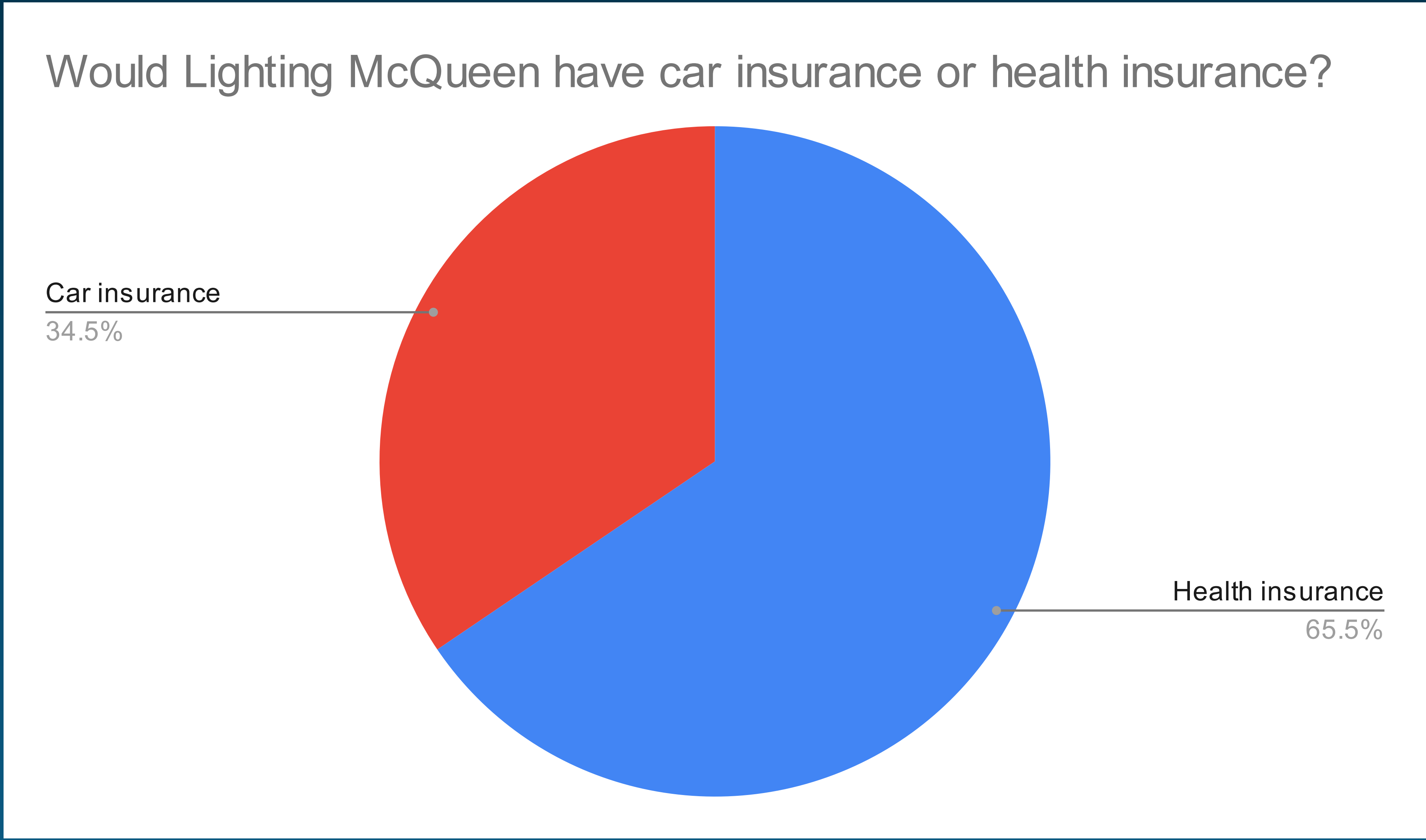
- ANTS  
    - Checkpoint 2 due today 03/21
    - The whole project due Fri 03/24
    - Submit by Thu 03/23 for one extra point!
- Lab 09 due tomorrow 03/22
- Come to OH >:)

ABOUT THE 2ND MIDTERM 😮

The second… what?

- Fri 4/7, 7-9 pm
- Logistics - Ed post #2141
  - If you need ANY alterations (left-handed desk, remote, other accommodations due to DSP or otherwise), fill out this form by Mon 04/03!!
- Preparations
  - Familiarize yourself with the topics in scope
  - Attend review session (or watch recordings/slides) for more topical review - see Ed for more info
  - Do practice exams!
    - Quality > quantity
    - Post on exam threads on Ed for help
    - Walkthrough videos/guide are your friend!

FROM LAST TIME... 👀



MUTABLE TREES 🌳

WHY MUTABLE TREES?

Recall from (functional) data abstraction...

```
def tree(label, branches=[]):
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
def is_leaf(tree):
    return not branches(tree)
```

- These trees are immutable
  - To modify, need to call constructor again with updated attributes - inefficient 😿
  - Solution - use OOP, since objects are mutable 😻

MUTABLE TREES

```python
class Tree:
    def __init__(self, label, branches=[]):
        for b in branches:  # branches should be a list of trees
            assert isinstance(b, Tree)
        self.label = label
        self.branches = branches
    def is_leaf(self):
        return not self.branches
```

```python
>>> t = Tree(3, [Tree(2, [Tree(5)]), Tree(4)])
>>> t.label # label is now an instance attribute
3
>>> t.branches[0].label # so is branches
2
>>> t.branches[1].is_leaf() # is_leaf is a method
True
```

FUNCTIONAL DATA ABSTRACTION VS. OOP

	Functional Data Abstraction	Tree class / OOP
Constructor	`tree(label, branches)`	`Tree(label, branches)` (which calls `Tree.__init__`)
Label and branches	`label(t)` or `branches(t)`	`t.label` or `t.branches`
Mutability	immutable - cannot assign values to call expressions	mutable - can reassign `label` and `branches`
Checking if a tree is a leaf	convenience function `is_leaf(t)`	bound method `t.is_leaf()`

TREES - PROBLEM SOLVING STRATEGIES

- Pay attention to whether it's **mutation** or **constructing a new tree**
- To mutate a tree object:
  - reassign its instance attributes (`t.label = ...` or `t.branches = ...`)
  - use list mutation method (append/extend/pop/etc.) on its branches - `t.branches` is a **list** of trees!
- For mutation problems:
  - Which should be mutated first - the root node or its branches?
  - The return value is often `None`
  - Sometimes the case case is implicit - if we have for loop that iterates through all the branches, the loop will not be executed if `t.branches` is an empty list (i.e., when `t` is a leaf)

LAB Q1

ORDER OF GROWTH

ORDER OF GROWTH

- Order of growth (efficiency) - how the runtime changes as the input size increases
  - Think of runtime as a <u>**function**</u> of the input size
- Input size (not the exact definition, but as a rule of thumb)
  - numeric input - magnitude of the number
  - Python lists - length of the list
  - linked list/trees/other recursive objects - number of nodes
- Runtime (not the exact definition, but as a rule of thumb)
  - the number of operations
- Theta notation - for input of size n, the runtime is denoted by $\Theta(f(n))$
  - $\Theta(f(n))$ - approximately $f(n)$ by a constant factor

ORDER OF GROWTH - OTHER NOTES

- constant < logarithmic < linear < quadratic < exponential
- Constants are ignored
  - E.g., $\Theta(2n + 3)$ is essentially $\Theta(n)$
- Only consider the term that grows fastest
  - E.g., $\Theta(n^2 + 2n + 3)$ is essentially $\Theta(n^2)$

ORDER OF GROWTH - CONSTANT

- Constant ↔ $\Theta(1)$
- Runtime does not change as the input size changes
- For example:

```
def square (x):
    return x * x
```

input	function call	return value	operations
1	`square(1)`	1*1	1
2	`square(2)`	2*2	1
...	...	...	...
100	`square(100)`	100*100	1
...	...	...	...
n	`square(n)`	n*n	1

ORDER OF GROWTH - LOGARITHMIC

- Logarithmic $\leftrightarrow$ $\Theta(\log n)$
- Often when we keep dividing the input by a constant

```
def foo(x):
    while x > 0:
        print('hey')
        x //= 2
```

Let's say the while loop runs n times before x reaches 0.

divide x by 2 for n times before it gets to 1

$$\implies \frac{x}{2^n} = 1 \implies 2^n = x \implies n = \log_2 x$$

ORDER OF GROWTH - LINEAR

- Linear $\leftrightarrow \Theta(n)$
- Often when a loop runs n times, each time doing work in constant time

```python
def factorial(x):
    prod = 1
    for i in range(1, x + 1):
        prod *= i # execute x times in total
    return prod
```

input	function call	return value	operations
1	`factorial(1)`	1*1	1
2	`factorial(2)`	2*1*1	2
...	...	...	...
100	`factorial(100)`	100*99*...*1*1	100
...	...	...	...
n	`factorial(n)`	n*(n-1)*...*1*1	n

ORDER OF GROWTH - QUADRATIC

- Quadratic $\leftrightarrow \Theta(n^2)$
- Often when a nested loop runs n^2 times, each time doing work in constant time

```python
def bar(n):
    for a in range(n):
        for b in range(n):
            print(a, b)  # print n * n times in total
```

input	function call	operations (prints)
1	`bar(1)`	1
2	`bar(2)`	4
...	...	...
100	`bar(100)`	10000
...	...	...
n	`bar(n)`	n^2

ORDER OF GROWTH - EXPONENTIAL

- Exponential $\leftrightarrow \Theta(c^n)$, where c is a constant
- Often in tree recursion

```
def rec(n):
    if n == 0:
        return 1
    else:
        return rec(n - 1) + rec(n - 1)
```

input	function call	return value	operations
1	`rec(1)`	2	1
2	`rec(2)`	4	3
...	...	...	...
10	`rec(10)`	1024	1023
...	...	...	...
n	`rec(n)`	`2^n`	`2^n`

ORDER OF GROWTH - TREE RECURSION

```python
def rec(n):
    if n == 0:
        return 1
    else:
        return rec(n - 1) + rec(n - 1)
```

- Try drawing out the recursion tree diagram
- $n + 1$ levels
- The i^{th} level has 2^i nodes (root at level 0)
- Each node does constant work (addition)
- Total = $(1 + 2 + \ldots + 2^{n+1}) \cdot constant = \Theta(2^n)$

ORDER OF GROWTH - TREE RECURSION

What about this? 🤔

```
def rec(n):
    if n == 0:
        return 1
    else:
        return rec(n // 2) + rec(n // 2)
```

- Draw out the recursion tree diagram
- $\log_2 n$ levels
- The i^{th} level has 2^i nodes (root at level 0)
- Each node does constant work (addition)
- Total = $(1 + 2 + \ldots + 2^{\log_2 n}) \cdot constant = \Theta(n)$

ORDER OF GROWTH - NESTED LOOPS

- In general, (# times the loops run) * (work done each time)

```
def factorial(n):
    # returns n! in linear time
    ...

def foo(n):
    # nested loop runs n^2 times
    for i in range(n):
        for j in range(n):
            # each time this takes theta(n) time
            print(factorial(n))
```

- `foo(n)` runs in $\Theta(n^3)$ time

LAB Q5

NOW IT'S YOUR TIME 🤠

- Get started on the lab and raise your hand whenever you need help!
- Get to know your neighbors and collaborate if you'd like!
- Slides: go.cs61a.org/mingxiao-index
- Leave any anonymous feedback here: go.cs61a.org/mingxiao-anon

AND REMEMBER TO GET CHECKED OFF! 👒

---

go.cs61a.org/mingxiao-att

The secret phrase is ...
(NOT 3 dots! I'll announce it 🙉)