M269 Prsntn2019J TMA02

Question Review

Phil Molyneux

22 March 2020

TMA02 Q3(a)(i)

Question

- State and justify the Big-O complexities of
- `has_node`
- `has_edge`
- `nodes`
- `edges`
- `common`
- Assume a graph has `n` nodes and `e` edges
- Take assignments as the basic unit of computation
- For set and dictionary operations, use the complexities given in the Companion's appendix.

TMA02 Q3(a)(i)

Code (1)

```
def __init__(self):
  self.graph = dict()

def has_node(self, node):
  return node in self.graph

def has_edge(self, node1, node2):
  return (self.has_node(node1)
          and node2 in self.graph[node1])
```

- Code from `TMA02_Q3_graph.py` with comments removed and slightly edited for layout
- Type of a `Graph`
- Type and operations of `has_node` :
- Type and operations of `has_edge` :

TMA02 Q3(a)(i)

Code 1(a)

- `Graph` is a dictionary with `key` is a `node` (a string) and the `value` is the set of the node's neighbours
- `has_node` takes a `Graph` and a `node` and returns a boolean
- Operations: dictionary `in`
- `has_edge` takes a `Graph` and two `node` items and returns a boolean
- Operations: three operations: `has_node` and two dictionary operations `in` and *get value*

TMA02 Q3(a)(i)

Code (2)

```
def nodes(self):
    result = set()
    for node in self.graph:
        result.add(node)
    return result

def edges(self):
    result = set()
    for node1 in self.nodes():
        for node2 in self.nodes():
            if self.has_edge(node1, node2):
                if (node2, node1) not in result:
                    result.add((node1, node2))
    return result
```

- Code from `TMA02_Q3_graph.py` with comments removed and slightly edited for layout
- Type and operations of `nodes` :
- Type and operations of `edges` :

TMA02 Q3(a)(i)

Code 2(a)

- `nodes` takes a `Graph` and returns a set of `node` items
- Operations: adds each of the `n` nodes to the output set
- `edges` takes a `Graph` and returns a set of `node` pairs
- Operations: double loop over all possible pairs of nodes

TMA02 Q3(a)(i)

Code (3)

```
def common(friendships, person1, person2):
  assert person1 != person2
  assert person1 in friendships.nodes()
  assert person2 in friendships.nodes()

  mutual = 0
  for person in friendships.nodes():
    if (friendships.has_edge(person, person1) and
      friendships.has_edge(person, person2)):
      mutual = mutual + 1
  return mutual
```

- Code from `TMA02_Q3.py` with comments removed and slightly edited for layout
- Type and operations of `common` :

TMA02 Q3(a)(i)

Code 3(a)

- `common` takes a `Graph` and two `persons` (strings) and returns `mutual`, an integer
- Operations: at most three operations per node, one loop and at most two `has_edge` operations

TMA02 Q3(a)(i)

Hints

- Check the operations for each part
- `has_node`
- `has_edge`
- `nodes`
- `edges`
- `common`

Time Complexities of Collection Types

M269 Python

- The next two slides are from

  M269TutorialBinaryTrees2019J.beamer.pdf slide 327/337

  M269Prsntn2019JTutorialProgPythonADT.beamer.pdf slide 78/171

Sets, Maps

Implementation Points

	Python		Haskell
Operation	**Average**	**Worst**	**Worst**
Member	$O(1)$	$O(n)$	$O(\log n)$
Union	$O(m+n)$		$O(m \log(\frac{n}{m}+1))$
Intersection	$O(\min(m,n))$	$O(m \times n)$	$O(m \log(\frac{n}{m}+1))$
Difference	$O(m)$		$O(m \log(\frac{n}{m}+1))$
Insert	$O(1)$	$O(n)$	$O(\log n)$
Delete	$O(1)$	$O(n)$	$O(\log n)$

- Python: Time Complexity
- Haskell: Data.Set and Data.Map.Strict
- Remember that actual behaviour will depend on the data and compiler settings

Program Complexity

Python Data Types — Lists

Operation	Notation	Average	Amortized Worst
Get item	`x = xs[i]`	$O(1)$	$O(1)$
Set item	`xs[i] = x`	$O(1)$	$O(1)$
Append	`xs = ys + zs`	$O(1)$	$O(1)$
Copy	`xs = ys[:]`	$O(n)$	$O(n)$
Pop last	`xs.pop()`	$O(1)$	$O(1)$
Pop other	`xs.pop(i)`	$O(k)$	$O(k)$
Insert(i,x)	`xs[i:i] = [x]`	$O(n)$	$O(n)$
Delete item	`del xs[i:i+1]`	$O(n)$	$O(n)$
Get slice	`xs = ys[i:j]`	$O(k)$	$O(k)$
Set slice	`xs[i:j] = ys`	$O(k+n)$	$O(k+n)$
Delete slice	`xs[i:j] = []`	$O(n)$	$O(n)$
Member	`x in xs`	$O(n)$	
Get length	`n = len(xs)`	$O(1)$	$O(1)$
Count(x)	`n = xs.count(x)`	$O(n)$	$O(n)$

- Source https://wiki.python.org/moin/TimeComplexity
- See https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

M269 Python Collection Types

Notes on M269 Companion (1)

- The M269 *Companion* lists the Python built-in collections and operation that can e used in M269
- If complexities are different in best and worst cases the tables give $O(\ldots)/O(\ldots)$
- `xs` and `ys` are collections with `m` and `n` elements
- The basic **sequence types** are lists, tuples and range objects
- **Lists** are ordered collections of items of the same type accessed by index
- **Tuples** are sequences which are ordered but can be of heterogenous types and are immutable
- **Ranges** represent an immutable sequence of numbers — used for looping

M269 Python Collection Types

Notes on M269 Companion (2)

- **Sets** are unordered collections of distinct hashable objects
- The `set` type is mutable; the `frozenset` type is immutable and hashable
- A mapping object maps hashable values to arbitrary objects
- There is one mapping type, the *dictionary* with the `dict` constructor
- A dictionary's keys must be hashable

M269 Python Collection Types

Lists

Operation	Notation	Complexity
Append item	`xs.append(x)`	$O(1)$
Comparison	`xs == ys`	$O(1)/O(n)$
Append list	`xs = ys + zs`	$O(m+n)$
Create list	`[]`, `list()`	$O(1)$
Get item	`x = xs[i]`	$O(1)$
Set item	`xs[i] = x`	$O(1)$
Get length	`n = len(xs)`	$O(1)$
Member	`x in xs`	$O(1)/O(n)$
Pop last	`xs.pop()`	$O(1)$
Pop other	`xs.pop(i)`	$O(k)$
Get slice	`xs = ys[i:j]`	$O(k)$ / $O(k)$
Copy	`xs = ys[:]`, `xs = ys.copy()`	$O(n)$
Insert(i,x)	`xs[i:i] = [x]`	$O(n)$
Delete item	`del xs[i:i+1]`	$O(n)$
Set slice	`xs[i:j] = ys`	$O(k+n)$
Delete slice	`xs[i:j] = []`	$O(n)$
Count(x)	`n = xs.count(x)`	$O(n)$
Sort in-place	`xs.sort()`	$O(n)$ / $O(n \log n)$
Sort	`ys = sorted(xs)`	$O(n)$ / $O(n \log n)$

M269 Python Collection Types

Sets

Operation	Notation	Complexity
Add item	`xs.add(x)`	$O(1)$
Size	`len(xs)`	$O(1)$
Copy	`xs = ys.copy()`	$O(n)$
Difference	`xs - ys`, `xs.difference(ys)`	$O(n)$
Empty set	`set()`	$O(1)$
Intersection	`xs & ys`, `xs.intersection(ys)`	$O(n)$
Member	`x in xs`	$O(1)$ / $O(n)$
Delete existing item	`xs.remove(x)`	$O(1)$
Delete item	`xs.discard(x)`	$O(1)$
Remove/return item	`xs.pop()`	$O(1)$
Subset	`xs <= ys`, `xs.issubset(ys)`	$O(n)$
Union	`xs | ys`, `xs.union(ys)`	$O(m+n)$
Sort	`ys = sorted(xs)`	$O(n)$ / $O(n \log n)$

M269 Python Collection Types

Dictionary

Operation	Notation	Complexity
Add or replace	`xs[key] = value`	$O(1)$ / $O(n)$
Size	`len(xs)`	$O(1)$
Empty dictionary	`dict()`, `{}`	$O(1)$
Copy	`xs = ys.copy()`	$O(n)$
Get value	`x = xs[key]`	$O(1)$ / $O(n)$
Member	`x in xs`	$O(1)$ / $O(n)$
Delete existing item	`del xs[key]`	$O(1)$ / $O(n)$
Sort	`ys = sorted(xs)`	$O(n)$ / $O(n \log n)$

TMA02 Q3(a)(ii)

Question

- Initial insight for the given `common` code
- Initialise `mutual` to `0`
- For each node in `friendships` :
- If that node has edges to `person1` and `person2` then it is a mutual friend and hence increment `mutual`
- Give an initial insight for a more efficient algorithm for `common`.
- Use only operations in the `Graph` class or in the *Companion's* appendix

TMA02 Q3(a)(ii)

Hints

- What were the operations for `common` in the original form ?
- What alternatives are there which use fewer operations ?
- How can we use other functions, such as `neighbours` or `friends_of_friends`, `suggested_friends` ?

TMA02 Q3(a)(ii)

Hints

- What were the operations for `common` in the original form ?
- What alternatives are there which use fewer operations ?

TMA02 Q3(a)(ii)

Code (1)

```
def neighbours(self, node):
  assert self.has_node(node)

  # copy the set of neighbours
  result = set()
  for neighbour in self.graph[node]:
    result.add(neighbour)
  return result
```

- Defined in TMA02_Q3_graph.py
- Essentially making a copy of the set of neighbours
- Why is it doing this ?

TMA02 Q3(a)(ii)

Code (2)

```
def friends_of_friends(friendships, person):
  assert person in friendships.nodes()

  result = set()
  people = friendships.nodes()
  for person1 in people:
    for person2 in people:
      if (friendships.has_edge(person1, person2) and
        friendships.has_edge(person2, person) and
        not friendships.has_edge(person1, person) and
        person1 != person):
        result.add(person1)
  return result
```

- Returns set of friends of friends
- Notice the explicit line joining

TMA02 Q3(a)(ii)

Code (3)

```
def suggested_friends(friendships, person):
  assert person in friendships.nodes()

  scored_suggestions = []
  for friend_of_friend in friends_of_friends(friendships, person):
    score = common(friendships, person, friend_of_friend)
    scored_suggestions.append((score, friend_of_friend))
  scored_suggestions.sort(reverse=True)
  suggestions = []
  for (score, suggestion) in scored_suggestions:
    suggestions.append(suggestion)
  return suggestions
```

- Returns list of suggested friends
- List is ordered by number of mutual friends
- Cost ?

TMA02 Q3(a)(iii)

Question

- Justify your answer being more efficient for the typical case in which every person has some friends but is not friends with everyone else.
- You are not required to provide a Big-O analysis.

TMA02 Q3(a)(iii)

Hints

- I what way does just looking at `neighbours` reduce the comparisons ?

TMA02 Q3(a)(iv)

Question

- Produce a fresh version of `common` with code following your initial insight
- Use only operations in the `Graph` class or in the *Companion's* appendix

TMA02 Q3(a)(iv)

Hints

- Loop through `person1` neighbours, checking if in `person2` neighbours
- Use the set intersection operator or function

TMA02 Q3(a)(v)

Question

- A client wants to tell a user how to be introduced to another user (if it is possible)
- Explain what graph algorithm you might adopt (or adapt)
- Do not provide code
- You may use any graph algorithm mentioned in Unit 5

TMA02 Q3(a)(v)

Hints

- What algorithms are mentioned in Unit 5 ?
- Dijkstra's algorithm
- Breadth-first search
- Depth-first search
- See the *M269 Companion* on *BFS and DFS* and *Distance problems*
- How could you adapt each ?
- And what are the pros and cons ?

M269 TMA02

Summary

- The assignment investigates *choices* between different data structures and algorithms
- No single correct answer but simple, easier-to-understand algorithms are a better starting point
- Should (in a future course) looks for ways of *transforming* simple, but correct, algorithms into more efficient ones using various useful laws of computation

Future Work

Topics & Events

- Wednesday, 25 March 2020 TMA02 due
- Sunday, 5 April 2020 online tutorial Unit 6 Logic
- Wednesday 29 April 2020 iCMA46 due
- Sunday, 3 May 2020 online tutorial Unit 7 Computability, Complexity
- Sunday, 17 May 2020 online tutorial exam revision
- Saturday, 23 May 2020 tutorial at the LSE Aldwych exam revision
- Wednesday 27 May 2020 iCMA47 due
- Thursday 11 June 2020 Exam
- Please email me with any requests for particular topics

Web Sites

Sorting

- **Big-O Cheat Sheet** http://bigocheatsheet.com/

Web Sites

Graph Algorithms

- **Graph Theory** http://en.wikipedia.org/wiki/Graph_theory
- **Tree (Graph Theory)** http://en.wikipedia.org/wiki/Tree_(graph_theory)
- **Dekai Wu Algorithms course** https://www.cse.ust.hk/~dekai/271/
- **Jeff Erickson Algorithms** http://jeffe.cs.illinois.edu/teaching/algorithms/

Web Sites

Dynamic Programming

- *Jeff Erickson Algorithms* http://jeffe.cs.illinois.edu/teaching/algorithms/
- *Cormen* (2009, page 407)has same example as *Jeff Erickson*
- *Peter Norvig Spell Checker*http://norvig.com/spell-correct.html (from http://stackoverflow.com/questions/2460177/edit-distance-in-python)
- *Rosetta Code: Levenshtein Distance* http://rosettacode.org/wiki/Levenshtein_distance
- *Edit Distance HaskellWiki* https://wiki.haskell.org/Edit_distance
- *Edit Distance in Haskell* http://stackoverflow.com/questions/5515025/edit-distance-algorithm-in-haskell-performance-tuning
- *Wikibooks Algorithm implementation — Levenshtein Distance*