CSC148 L5102
Introduction to Computer Science
Joe Lim
joe.lim@utoronto.ca
Office Hours: 5-6pm PB150
http://www.cdf.toronto.edu/~csc148h/winter/

Reminders

- Course Evaluation
- Final Exam:

<table>
<thead>
<tr>
<th>Section</th>
<th>A</th>
<th>Tue, 9-11</th>
<th>Ex 7:00-8:00</th>
<th>Ex 9:00-10:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section</td>
<td>B</td>
<td>Tue, 9-11</td>
<td>Ex 7:00-8:00</td>
<td>Ex 9:00-10:00</td>
</tr>
<tr>
<td>Section</td>
<td>C</td>
<td>Tue, 11-1</td>
<td>Ex 7:00-8:00</td>
<td>Ex 9:00-10:00</td>
</tr>
</tbody>
</table>

Outline
- Hash Tables
- Tracing Code
- Review

Why hash

Lists are contiguous (adjacent) sequences of references to objects, so access to a list position is fast (just arithmetic)

What if we would convert - hash - other data to a suitable integer for a list index, we'd want:
- Fast
- Deterministic: the same (or equivalent values) gets hashed to the same integer each time
- Well-distributed: We'd like a typical set of values to get hashed pretty uniformly over the available list positions

You can't hash everything!

```python
>>> list1 = [0]
>>> id(list1)
3069263116
>>> list2 = [0, 1]
>>> id(list2)
3069528300
>>> list1.append(1)
>>> id(list1)
3069263116
Oops!
```

Hash to hash table (dictionary)...

Once you have hashed an object to a number, you can easily use part of that number as an index into a list to store the object, or something related to that object. If the list is of length n, you might store information about object o at index hash(o) % n.
Collisions

- Even a well-distributed hash function will have a surprising number of collisions...
- How many people do you need to poll before you find two with the same birthday (out of 366 possibilities, including leap-year)?
- The mathematics is a bit counter-intuitive... the probability of a non-collision for 23 birthdays is:

\[ p = \frac{365}{366} \times \frac{364}{366} \times \ldots \times \frac{344}{366} \approx 0.493 \]

Chaining or probing

A couple of tactics for dealing with two different keys ending up at the same index:
- Chaining: keep a small (one hopes) list at the index
- Probing: explore, in a systematic way, until the next open index

Either tactic has costs, so keep collisions to a minimum by keeping the list partly empty

Python dictionaries are implemented using hash tables and probing. The cost of collisions is kept small by enlarging the underlying table when necessary, and the cost of enlarging is amortized over many dictionary accesses.

The result is the access to a dictionary element is \(\mathcal{O}(1)\), essentially the time it takes to access a list element.

One downside is that extra work is required to order the keys or values of a dictionary. What is their "natural" order?

Know Your Code

... inside out, left to right

```python
def f(n):
    return n + 1
def g(m):
    return f(m) + 1
print(f(g(f(1) + 2) + 2))
```

Know More Code

... bottom to top

```python
class A:
def g(self, n):
    return n + 1
def f(self, m):
    return self.g(m)
class B(A):
def g(self, n):
    return 2 * n
if __name__ == '__main__':
b = B()
print(b.f(2))
a = A()
print(a.f(2))
```

... and more code

... think locally...

```python
x = 7
def f():
    y = x
    print(y)
    if False:
        x = 2
    if name_ == "main_":
        f()
```
Persistent values

Default values are created when a function is defined...

```python
>>> def f(n, m=[]):
...     m.append(n)
...     return m
...>>> f(1)
[1]
>>> f(2)
[1, 2]
```

Develop a function that will merge two linked list in the following manner:

L1: 1 -> 2 -> 3 -> 4 -> 5
L2: 10 -> 20 -> 30 -> 40

merge (L1, L2) would result into:
L1: 1 -> 40 -> 2 -> 30 -> 3 -> 20 -> 4 -> 10 -> 5
L2: 10 -> 20 -> 30 -> 40 (unchanged)

L3: 3 -> 6 -> 9
L4: 12 -> 15 -> 18 -> 21

merge (L3, L4) would result into:
L3: 3 -> 21 -> 6 -> 18 -> 9 -> 15 -> 12
L4: 12 -> 15 -> 18 -> 21 (unchanged)

For y in range(n):
    print (y)
for z in range(y):
    print (z)

```python
for y in range(n):
    print (y)
for z in range(y):
    print (z)
```
What is the Big-Oh Notation?

for i in range(n, 0, i /= 2):
    for j in range(1, n, j *= 2):
        #...constant number of operations

Yet another example of Big-Oh

for bound in range(1, n + 1, bound *= 2):
    for i in range(0, bound, i++):
        for j in range(0, n, j += 2):
            ...

Pell numbers are an infinite sequence of integers. It can be expressed by the formula:

\[ P_n = \frac{(1 + \sqrt{2})^n + (1 - \sqrt{2})^n}{2\sqrt{2}} \]

Create a recursive function to calculate the Pell number of n. What is the big-oh notation \( \Omega() \) for your function?

For each node in a binary search tree, create a new duplicate node, and insert the duplicate as the left child of the original node. The resulting tree should still be a binary search tree.

```
def doubleTree(node):
    #...description
```

We'll define a "root-to-leaf path" to be a sequence of nodes in a tree starting with the root node and proceeding downward, but a path is not considered if an empty tree contains no root-to-leaf paths. So for example, the following tree has exactly four root-to-leaf paths:

```
5 /\ 4
/ \ / \ 11
1 2
```

Root-to-leaf paths:
- path 1: 5 4 11 7
- path 2: 5 4 11 2
- path 3: 5 8 13
- path 4: 5 8 4

For this problem, we will be concerned with the sum of the values of such a path — for example, the sum of the values on the 5-4-11-7 path is 5 + 4 + 11 + 7 = 27.

Given a binary tree and a sum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. Return false if no such path can be found.

```
def hasPathSum(node, sum):
    """Return true if the tree has a root-to-leaf path equals the given sum."""
```