Algorithms and Data Structures for Data Science
Arrays and Asymptotic Efficiency

CS 277
Brad Solomon

February 6, 2023
Exam 1

Exams will be proctored by the CBTF: https://cbtf. engr. illinois. edu/

(That link will have a link to Prairietest, where you can sign up for exam 1)

Reservations open on February 2nd @ 9 AM

You must take the exam sometime between 2/14 and 2/16!

See website for expected content:

https://courses.grainger.illinois.edu/cs277/sp2023/exams/
Learning Objectives

Conceptualize the array list implementation

Introduce the concept of asymptotic efficiency

Compare list implementations using big O
List Implementations

1. Linked List

2. Array List
Creating an array list

```python
1  myList = []
2  l = [0]*5
3  matrix = numpy.zeros([2, 3])
```
import numpy as np

r1 = [7, 8, 9]
r2 = [4, 5, 6]
r3 = [1, 2, 3]
plist = [r1, r2, r3]

myNP = np.array(plist)

print(myNP)

myNP = myNP.reshape(1, 9)

print(myNP)
Array `__len__()`

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>len(l)</code></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>numpy.shape(matrix)</code></td>
</tr>
</tbody>
</table>
Array \_\_getitem\_\_()
Array `__getitem__()`

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>1[3]</code></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>1[9000]</code></td>
</tr>
</tbody>
</table>
PyObject *
PyList_GetItem(PyObject *op, Py_ssize_t i)
{
    if (!PyList_Check(op)) {
        PyErr_BadInternalCall();
        return NULL;
    }
    if (i < 0 || i >= Py_SIZE(op)) {
        if (indexerr == NULL)
            indexerr = PyString_FromString(
                "list index out of range");
        PyErr_SetObject(PyExc_IndexError, indexerr);
        return NULL;
    }
    return ((PyListObject *)op)->ob_item[i];
}
Array ‘Add’

1. append(1)
2.
3. insert(0, 2)
Array Capacity vs Array Size

We want to minimize the number of times we have to rebuild an array!
Resize Strategy: +2 elements every resize
Resize Strategy: +2 elements every resize
Resize Strategy: x2 elements every resize
Resize Strategy: x2 elements every resize
memory_size = {}
for length in range(50):
    lst = []
    for length_loop in range(length):
        lst.append(length_loop)
    memory_size[length] = sys.getsizeof(lst)
print(memory_size)

```python
memory_size = {}
for length in range(50):
    lst = []
    for length_loop in range(length):
        lst.append(length_loop)
    memory_size[length] = sys.getsizeof(lst)
print(memory_size)
```

Trivia: Numpy append is really bad!

```python
nms = {}
for length in range(50):
    npa = np.array([])
    for length_loop in range(length):
        npa = np.append(npa, length)
    nms[length] = sys.getsizeof(npa)
print(nms)
```

```plaintext
```
Array remove()

```
1 = [1, 2, 3, 4, 5, 6, 7]
1.pop()
1.remove(1)
1.pop(3)
```
Python List Remove Implementation

```python
test = [1, 2, 3, 4, 5]
print(test, sys.getsizeof(test))
test.remove(3)
print(test, sys.getsizeof(test))
test.remove(4)
print(test, sys.getsizeof(test))
```

```
[1, 2, 3, 4, 5] 104
[1, 2, 4, 5] 104
[1, 2, 5] 104
```
List Abstract Data Type

We’ve now seen the LinkedList and Array versions of:

**Constructor:**

\[ \texttt{\_init\_()} \]

**Insert:**

\[ \texttt{append(x)} \quad \texttt{insert(i, x)} \]

**Delete:**

\[ \texttt{remove(x)} \quad \texttt{pop()} \]

**Index**

\[ \texttt{\_getitem\_()} \quad \texttt{index(x)} \]

**Size()**

\[ \texttt{len(list)} \]
Which implementation is better?
What do we care about when we write code?
How do we analyze an algorithm?

```python
for i in range(n):
    time.sleep(30)
    doStuff()
```

```python
for i in range(n):
    for j in range(n):
        doStuff()
```
How do we analyze an algorithm?

```
1 for i in range(n):
2   for j in range(n/2):
3      doStuff()
```

```
1 for i in range(n):
2   for j in range(n):
3      doStuff()
```
How do we analyze an algorithm?

$P$: word

$T$: There would have been a time for such a word

---word---word---word

$\Rightarrow$ $\Rightarrow$ $\Rightarrow$
How do we analyze an algorithm?

$p$: word

$t$: There would have been a time for such a word word word word word word word word word ...
    word word word word word word word word word word word word word word word word word ...
    word word word word word word word word word word word word word word word word word ...
    word word word word word word word word word word word word word word word word word ...
How do we analyze an algorithm?

\[ P: \text{aaa} \]
\[ T: \text{bbbbbbbb} \]

\[ P: \text{bbb} \]
\[ T: \text{bbbbbbbb} \]
How do we analyze an algorithm?
Big-O notation

\[ f(n) \text{ is } O(g(n)) \text{ iff } \exists c, k \text{ such that } f(n) \leq c g(n) \forall n > k \]
Constant Time, $O(1)$

```python
def constant(n):
    ops = 0
    for i in range(10):
        ops+=1
    return ops

print(constant(5))
print(constant(9001))
```
def logarithmic(n):
    ops = 0
    for i in range(int(math.log2(n))):
        ops+=1
    return ops

print(logarithmic(5))
print(logarithmic(9001))
Linear Time, $O(n)$

```python
def linear(n):
    ops = 0
    for i in range(n):
        ops+=1
    return ops

print(linear(5))
print(linear(9001))
```
# Quadratic Time

```python
# Quadratic Time
def quadratic(n):
    ops = 0
    for i in range(n):
        for j in range(n):
            ops+=1
    return ops

print(quadratic(5))
print(quadratic(9001))
```
Big-O Complexity Classes

- \(O(1)\)
- \(O(\log n)\)
- \(O(n)\)
- \(O(n^2)\)
- \(O(2^n)\)
Not every operation is created equal

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time x1 billion</th>
<th>Like</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5 seconds</td>
<td>Heartbeat 💓</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5 seconds</td>
<td>Yawn 😞</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 seconds</td>
<td>Long yawn 😞</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25 seconds</td>
<td>Make coffee ☕</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 seconds</td>
<td>Brush teeth</td>
</tr>
<tr>
<td>Compress 1K bytes</td>
<td>50 minutes</td>
<td>TV show 📺</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>5.5 hours</td>
<td>(Brief) Night's sleep 🛌</td>
</tr>
<tr>
<td>SSD random read</td>
<td>1.7 days</td>
<td>Weekend</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>2.9 days</td>
<td>Long weekend</td>
</tr>
<tr>
<td>Read 1 MB sequentially from SSD</td>
<td>11.6 days</td>
<td>2 weeks for delivery 📦</td>
</tr>
<tr>
<td>Disk seek</td>
<td>16.5 weeks</td>
<td>Semester</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>7.8 months</td>
<td>Human gestation 🐣</td>
</tr>
<tr>
<td>Above two together</td>
<td>1 year</td>
<td>🌍☀️</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>4.8 years</td>
<td>Ph.D. 🎓</td>
</tr>
</tbody>
</table>

(Care of [https://gist.github.com/hellerbarde/2843375](https://gist.github.com/hellerbarde/2843375))