

Introduction to Data Structures and Algorithms.

ENGF0002: Design and Professional Skills

Prof. Mark Handley, University College London

Term 1, 2019

Lists

Python Lists

```
days = ["mon", "tues", "weds", "thurs", "fri"]

#what's the second day of the week?
#recall, indices start at zero
day2 = days[1]  # "tues"

#add the weekend
days.append("sat")
days.append("sun")
# days is now ["mon", "tues", "weds", "thurs", "fri", "sat", "sun"]

#I don't like Mondays!
days.pop(0)
# days is now ["tues", "weds", "thurs", "fri", "sat", "sun"]

#We can remove from the end too
days.pop()
# days is now ["tues", "weds", "thurs", "fri", "sat"]
```

List Performance

How do Python lists perform?

How are Python lists actually implemented?

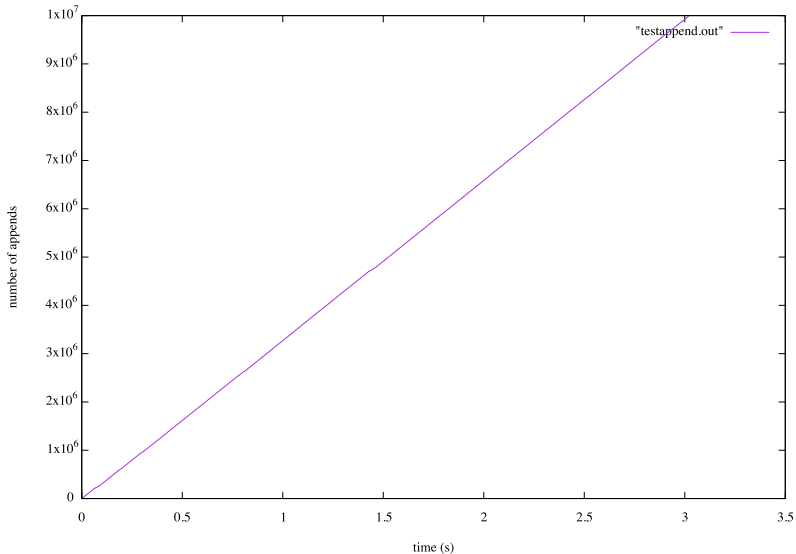
Python Lists: append performance

```
import time

lst = []
count = 0
starttime = time.time()
print(0, 0)
while count < 10000000:
    lst.append(count)
    if count % 10000 == 0:
        now = time.time()
        print(now - starttime, count)
    count += 1
```

Append a number to a list 10 million times. Print out elapsed time every 10,000 appends.

Python Lists: append performance



Python Lists: del performance

```
import time
lst = []
count = 0
while count < 10000000: #create a long list
    lst.append(count)
    count += 1

starttime = time.time()
count = 0
while count < 10000000: #delete all items, one by one
    lst.pop(0)
    if count % 10000 == 0:
        now = time.time()
        print(now - starttime, count)
    count += 1
```

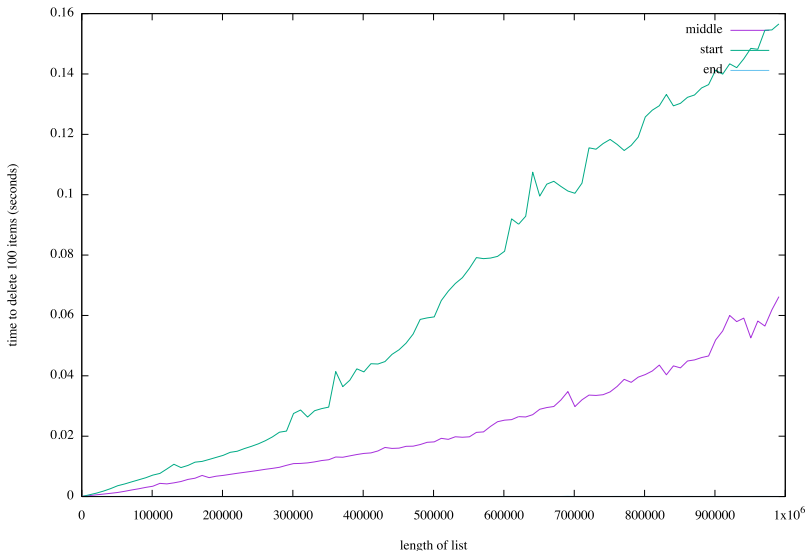
Create a 10 million item list. Delete front item, print out elapsed time every 10,000 deletions.

Python Lists: del performance

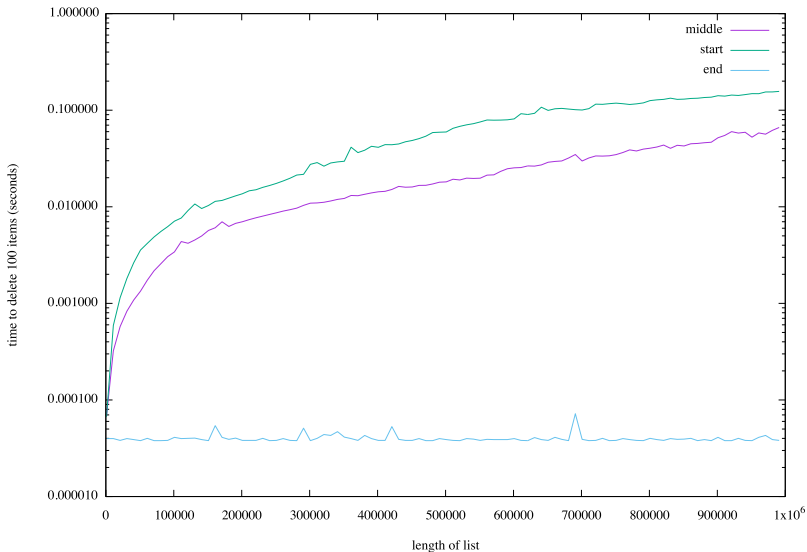
```
import time
lst = []
for length in range(1000, 1000000, 10000):
    while len(lst) < length:
        lst.append(42)
    count = 0
    starttime = time.time()
    while count < 200:
        lst.pop(0)           # pop from start
        #lst.pop()           # pop from end
        #lst.pop(length//2)  # pop from middle
        count += 1
    now = time.time()
    print(length, now - starttime)
```

Create a lists of increasing length. Time how long it takes to remove 200 items.

Python Lists: deletion performance



Python Lists: deletion performance, logscale

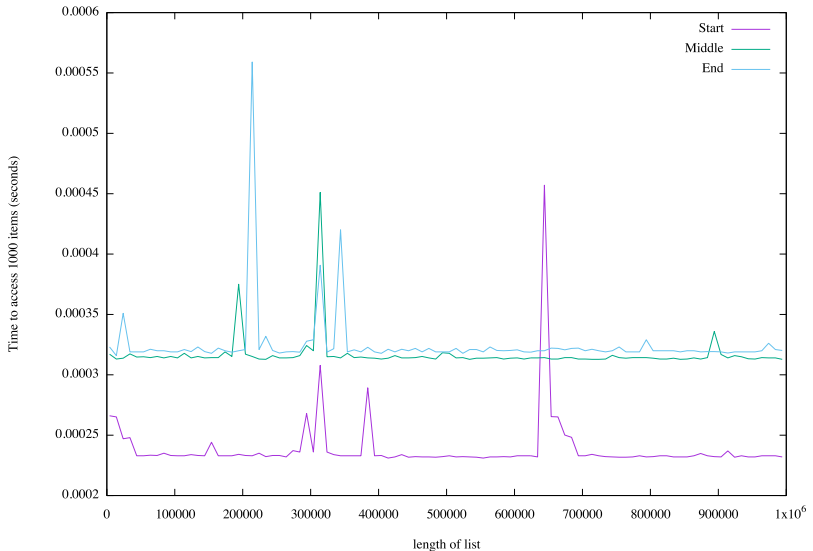


Python Lists: performance accessing items

```
import time
lst = []
for length in range(4000, 1000001, 10000):
    while len(lst) < length:
        lst.append(42)
    count = 0
    starttime = time.time()
    x = 0
    while count < 1000:
        x += lst[count]                # read from beginning
        #x += lst[length//2 + count]   # read from middle
        #x += lst[length - (count+1)]  # read from end
        count += 1
    now = time.time()
    print(length, now - starttime)
```

Create a lists of increasing length. Time how long it takes to access 1000 items.

Python Lists: performance accessing items



Profiling and Visualization

A profiler is software that analyzes what other software is doing.

Many different profilers exist; can profile different resources and at different levels.

Can we find out for sure where python is spending its time?

- A python profiler won't help - will tell us it's spending its time in pop()
- Need to profile the python interpreter itself.

Visualization

Profiling: dtrace

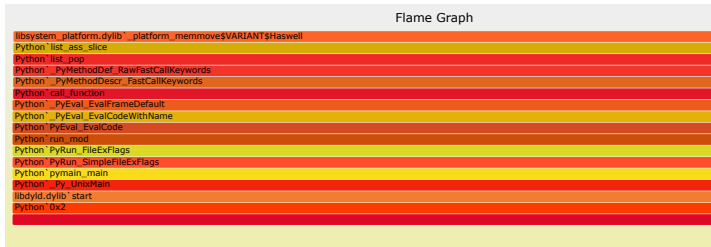
Advanced topic, you don't need to know this yet. Rough summary:

- I used dtrace to connect to the running python interpreter process.
- 1000 times per second, interrupted it, and logged what function was being called (known as a stack trace)
- Dumped the stats as a profile trace.
- Ran some fancy perl scripts to visualize the data.

me.svg

Flame graph of deletion from beginning

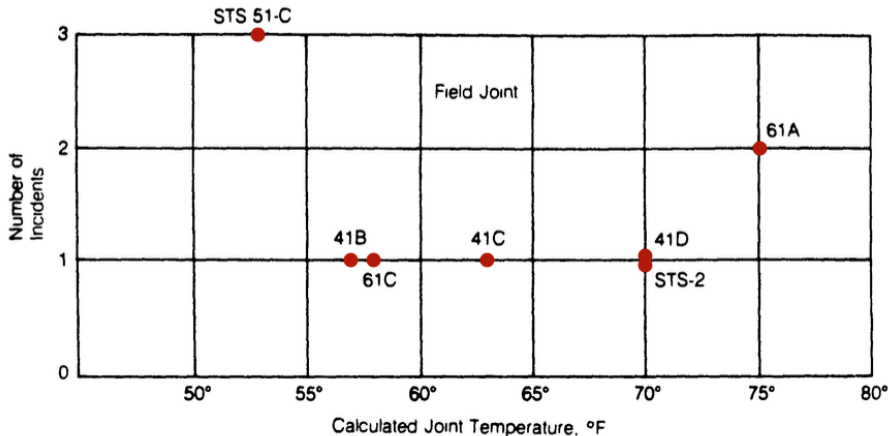
file:///Users/mjh/teaching/engf0002/engs102p_2018/misc/del3-flame.svg



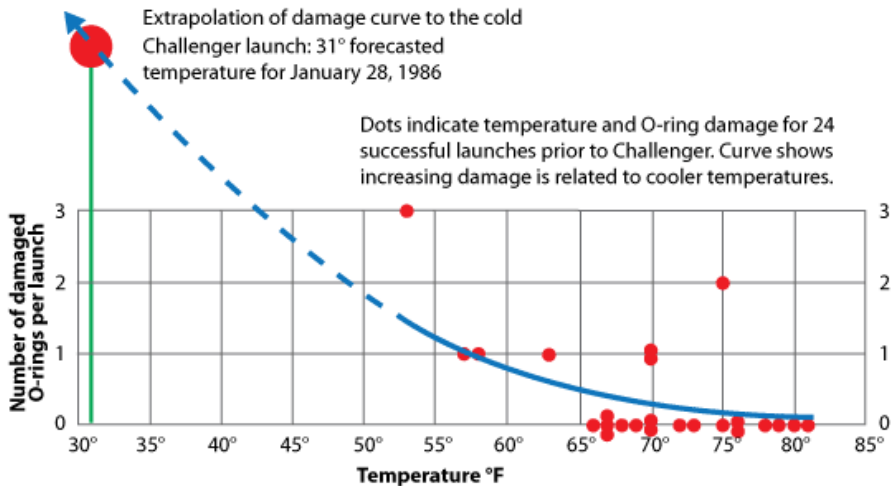
The Importance of Good Visualization



Graph from Thiokol's presentation to NASA



Edward Tufte's version of the same data



Algorithms & computational complexity

The Bookshop Problem



The Bookshop Problem: Solution



Best known algorithm: find Christina Foyle and ask her.

Is a value stored in a list?

Basically the same problem as Foyles bookshop.

We wish to define an algorithm `is_in` to test whether a list contains a target value.

- Return `True` if it is, and `False` otherwise.

Some **simple tests** for `is_in` would be:

```
def test_is_in():  
    assert is_in([1,2,3], 3)  
    assert not is_in([1,2,3], 4)
```

Sequential search.

```
def is_in(lst, target):  
    for value in lst:  
        if value == target:  
            return True  
    return False
```

- We construct a function that takes as parameters a list (lst) and target value (target).
- It uses a for loop to sequentially go through all the values of lst.
- Within the loop each value is tested. If they match return True.
- If the loop completes, it returns False.

How efficient is the `is_in` algorithm?

Assume the list has n elements:

- The algorithm will iterate over n elements if it returns `False`.
- If the target value is at a random position it will iterate on average over $\frac{1}{2}n$ elements.

The number of specific steps executed, as a function of the size of its inputs is a key measure of the **time complexity** of an algorithm.

However, we often only care about time complexity increases **within a constant factor** and for **large enough parameter sizes**.

The \mathcal{O} (Big Oh) notation.

Lets call the average number of steps an algorithm takes $f(n)$. We define as the **order of** relation between functions.

We say that $f(n) = \mathcal{O}(g(n))$ as $n \rightarrow \infty$ if

$$\exists n_0, c > 0. \quad |f(n)| \leq c \cdot |g(n)| \quad \text{for } n \geq n_0.$$

- In words, there exists a value n_0 after which the function $f(n)$ is always smaller than $g(x)$ up to a constant c .
- Note that the notation **hides lower order terms and constant**, eg. $3n^2 + 10n + 5 = \mathcal{O}(n^2)$.

Sequential search has time complexity in the order of $\mathcal{O}(n)$.

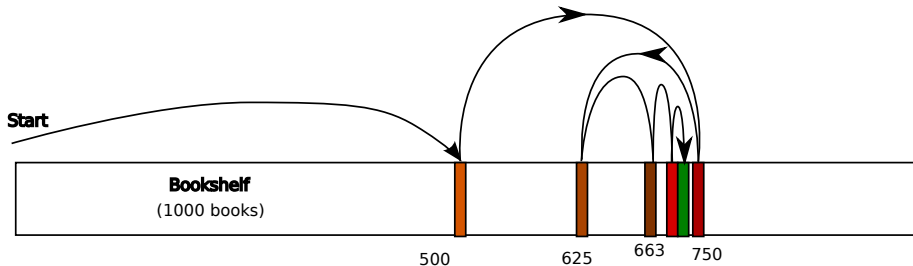
Can we search a sequence in fewer steps?

An arbitrary sequence cannot be searched in time less than $\mathcal{O}(n)$.

However, we can **pre-compute an index** on the sequence:

- An index is **a sorted view** of a sequence that allows fast **is_in** operations.









Binary search algorithm



How long does the search take?

Step	Books Eliminated	Books Remaining
1	500	500
2	750	250
3	875	125
4	938	62
5	969	31
6	984	16
7	992	8
8	996	4
9	998	2
10	999	1

The story of Sissa ben Dahir and the rice

							
1 grain	2 grains	4 grains	8 grains	16 grains	32 grains	64 grains	128 grains
256 grains	512 grains	1,024 grains	2,048 grains	4,096 grains	8,192 grains	16,386 grains	32,768 grains
4mm high	8mm high	16mm high	32 mm high	64mm high	13cm high	26cm high	51cm high
65,536	131,072	262,144	524,288	1,048,576	2,097,152	4,194,304	8,388,608
1m high	2m high	4m high	8m high	17m high	34m high	67m high	134m high
16,777,216	33,554,432	67,108,864	134,217,728	268,435,456	536,870,912	1,073,741,824	2,147,483,648
268m high	537m high	1km high	2km high	4km high	9km high	17km high	34km high
69km high	137km high	275km high	550km high	1100km high	2200km high	4400km high	8800km high
18,600km high	35,000km high	70,000km high	140,000km high	280,000km high	1.5x distance to moon	3x distance to moon	6x distance to moon
12x distance to moon	23x distance to moon	47x distance to moon	94x distance to moon	half way to sun	reaches the sun	2 AU	reaches Jupiter
reaches Saturn	almost reaches Uranus	reaches Neptune	8 light hours	17 light hours	1.4 light days	2.7 light days	5.6 light days

Binary search algorithm

- Define two variables `start` and `end`, initialized with the indexes of the first element and one past the last element
- Pick the middle element of the range.
 - if it is larger than the value sought, target is in lower half of the range. `end` takes the value of `middle`.
 - otherwise target is in upper half of range. `start` takes the value of `middle`
- Repeat until the range is of size one.
- If the element at the start of the range is the target value, return `True`.
- Otherwise, return `False`.

The code for binary search.

```
def is_in_bisect(lst, val):  
    """ Resturns True if val is within the sorted list lst. """  
    if len(lst) == 0:  
        return False  
    start = 0          # start of search range  
    end = len(lst)     # end of search range  
    while end - start > 1:  
        middle = (start + end) // 2  
        if val >= lst[middle]:  
            start = middle  
        else:  
            end = middle  
    return lst[start] == val
```

Binary search has $\mathcal{O}(\log n)$ time complexity.

Proof. Consider the size of the range $n = \text{end} - \text{start}$. $n = n_0$ at step 0. At each step the size of the range is a fraction $0 < \alpha < 1$ of its previous size, namely $n_i = \alpha \cdot n_{i-1}$. As we do integer division by 2, $\alpha \approx 0.5$

$$n_i = \alpha \cdot n_{i-1} \quad (1)$$

$$\Leftrightarrow n_i = \alpha^i \cdot n_0 \quad (2)$$

After how many steps i will n_i become 1, and the algorithm end?

$$\alpha^i \cdot n_0 = 1 \quad (3)$$

$$\Leftrightarrow \log(\alpha^i \cdot n_0) = 0 \quad (4)$$

$$\Leftrightarrow i \log \alpha + \log n_0 = 0 \quad (5)$$

$$\Leftrightarrow i = \left(-\frac{1}{\log \alpha} \right) \cdot \log n = \mathcal{O}(\log n). \quad (6)$$

The intimate link between data structures and algorithms.

Sequential search works on any sequence, but takes time $\mathcal{O}(n)$.

Binary search takes time $\mathcal{O}(\log n)$, but requires a sorted sequence. The data structure (sorted sequence) and the algorithm (binary search) work together and depend on each other.

Is $\mathcal{O}(\log n)$ really much better than $\mathcal{O}(n)$?

Remember $\mathcal{O}(\log n)$ is proportional to the number of binary digits in n (up to a constant). So if $n = 1000000$ then sequential search will take about a million steps, while binary search will take about 20. That is a big difference, and it grows as n grows.

Functions & Recursion.

A function is called **recursive**, or **using recursion**, if it is **calling itself**!

- A way of expressing a computation, possibly more naturally.
- Anything that can be done with recursion can be done with loops, and vice versa.
- Only a limited **recursive depth** is allowed.

What is recursion good for?

A number of problems are naturally expressed by dividing them into **sub-problems of the same form** as the initial problem. This algorithmic strategy is called **divide-and-conquer** and recursion is well suited to expressing a solution.

The recursive variant of binary search.

```
10 def isin_recursive(seq, val, start=0, end=None):
11     end = end if end is not None else len(seq)
12
13     if not end - start > 1:
14         return len(seq)>0 and (seq[start] == val)
15
16     mid = (start + end) // 2
17     if seq[mid] <= val:
18         return isin_recursive(seq, val, mid, end)
19     else:
20         return isin_recursive(seq, val, start, mid)
```

- Recursion at line 18 and 20.
- Other tricks: default parameters (line 10) and conditional expression (line 11).