A2 demo: if you don't have your mark posted, let me know the time and room.
A2 code: autotests have been run, the TAs should have it marked by Tuesday.

CSC148 winter 2017

(more...) efficiency

week 11

Danny Heap

heap@cs.toronto.edu / BA4270 (behind elevators)

http://www.cdf.toronto.edu/~csc148h/winter/

416-978-5899

March 29, 2017

Outline

big-Oh,Omega,Theta examples

hash tables

sequences

Binary digITS (BITS)

pretend that many operations (multiplication, division, comparison, etc.) are constant with respect to problem size (often called n), BUT for large enough n (man, many bits) this is not strictly true.

```
def silly(n):
    n = 17 * n**(1/2)
    n = n + 3
    print("n is: {}.".format(n))

    if n > 97:
        print('big!')
    else:
        print('not so big!')
```

constant wrt n

addition constant wrt n

constant wrt n

constant...

constant

How does the running time of **silly** depend on **n**?

constant

loops

How does the running time of this code fragment depend on **n**?

```
sum = 0
for i in range(n):
    sum += i
```

constant wrt

linear (proportional to n)
Theta(n)

proportional to n

i = 0, 1, 2, ..., n-1

constant wrt n

How does the running time of this code fragment depend on **n**?

```
sum = 0
for i in range(n//2):
    for j in range(n**2):
        sum += i * j
```

constant wrt

i = 0, 1, 2, ..., n//2 - 1

n//2 values

j = 0, 1, ..., n**2 - 1

n**2 values

(i, j) pairs? (n**3)//2

Theta(n**3)

Suppose it took x "steps" when n = 10
How many "steps" do you estimate when n = 30? --- 27x

more loops

How does the running of this code fragment depend on **n**?

```
i, sum = 0, 0
while i**2 < n:
    j = 0
    while j**2 < n:
        sum += i * j
        j += 1
    i += 1
```

constant wrt n

i = 0, 1, 2, ..., floor(sqrt(n) - 1) ... or sqrt(n)

constant wrt n

j = 0, 1, 2, 3, ..., floor(sqrt(n) - 1) ... or sqrt(n)

constant

n (i, j)

constant

constant

Theta(n)

How does the running time of this code fragment depend on **n**?

```
i, sum = 0, 0
while i < n * n:
    sum += i
    i += 1
```

conditions

How does the running time of this code fragment depend on **n**?

```
sum = 0
if n % 2 == 0:
    for i in range(n*n):
        sum += 1
else:
    for i in range(5, n+3):
        sum += i
```

O(n**2) # if n is even

Omega(n) # if n is odd

i = 0, ..., n**2 - 1

condition: you need to consider both branches

halving

How does the running time of **twoness** depend on **n**?

```
def twoness(n):
    count = 0
    while n > 1:
        n = n // 2
        count = count + 1
    return count
```

constant wrt n

n, n//2, n//4, n//8, ..., 1

floor(lg(n))

constant wrt

constant

constant

x for n

x+1 for 2n, x+2 for 4n, etc.

working with lg

$\lg(n)$: this is the number of times you can divide n in half before reaching 1.

- refresher: $a^b = c$ means $\log_a c = b$. definition of log
- this runtime behaviour often occurs when we "divide and conquer" a problem (e.g. binary search)
- we usually assume $\lg n$ (log base 2), but the difference is only a constant:

$$2^{\log_2 n} = n = 10^{\log_{10} n} \implies \log_2 n = \log_2 10 \times \log_{10} n$$

... so logs of different bases are related by a constant...

- so we just say $\mathcal{O}(\lg n)$.

miscellaneous

How does the running time of this code fragment depend on **n**?

constant --- n doesn't even appear in the code

```
for k in range(5000):
    if L[k] % 2 == 0:
        even += 1
    else:
        odd += 1
```

more miscellaneous

How does the running time of this code fragment depend on **n** and **m**?

```
sum = 0
for i in range(n):
    for j in range(m):
        sum += (i + j)
```

i = 0, 1, ..., n-1 -> n values

j = 0, 1, ..., m-1 -> m values

nxm (i, j) pairs...

summary

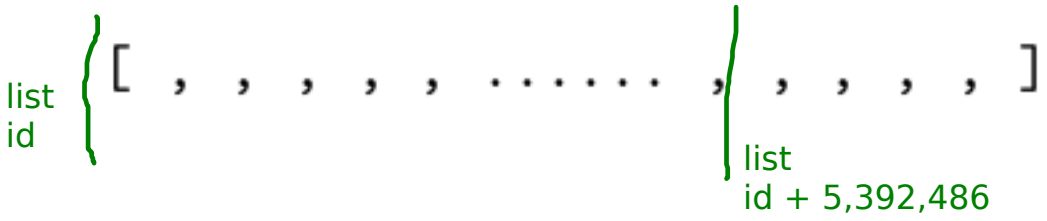
sequences: max

loops: count and multiply

conditions: consider each branch separately...

why hash

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



what if we could convert — hash — other data to a suitable integer for a list index, we'd want:

- fast — no sense being slow
- 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.

don't want every value to hash to ... 42

you can't hash everything!

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

oops! two equivalent objects should *not* have the same hash!

hash to hash table (dictionary)...

(key1, value1) , (key2, value2), ...

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 $\text{hash}(o) \% n$.

0 <= hash(o) < n

collisions

328, 338, 37, 280, 181, 114, 257, 200, 250, 68, 242, 250 ... 12 "birthdays" in class led to collision!

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:

$$p = \frac{366}{366} \times \frac{365}{366} \times \cdots \times \frac{344}{366} \approx 0.493$$

chaining or probing

a couple of tactics for dealing with two different keys ending up at the same index

hash("danny") % 8
hash("david") % 8 ...

"danny" "david"

```
[[], [], [], [], [], [], [], []]
```

"jacqueline"

chaining: keep a small (one hopes) list at that index (sometimes called bucket)

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[1] using hash tables and probing. The cost of collisions is kept small by enlarging the underlying list when necessary, and the cost of enlarging is amortized over many dictionary accesses.

The result is that access to a dictionary element is $\Theta(1)$, 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?

[1] Google "Tim Peters" but beware of obnoxious culture

Python dictionaries are implemented[1] using hash tables and probing. The cost of collisions is kept small by enlarging the underlying list when necessary, and the cost of enlarging is amortized over many dictionary accesses.

The result is that access to a dictionary element is $\Theta(1)$, 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?

[1] Google "Tim Peters" but beware of obnoxious culture