CSC148 winter 2016

more ... efficiency

week 11

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~heap/148/W16/
416-978-5899

March 30, 2016

Outline

big-Oh on paper

big-Oh examples

hash tables

quick sort

idea: break a list up (partition) into the part smaller than some value (pivot) and not smaller than that value, sort those parts, then recombine the list:

```
def qs(list_):
    """
    Return a new list consisting of the elements of list_ in
    ascending order.

    @param list list_: list of comparables
    @rtype: list

    >>> qs([1, 5, 3, 2])
    [1, 2, 3, 5]
    """
    if len(list_) < 2:
        return list_[:]
    else:
        return (qs([i for i in list_ if i < list_[0]]) +
                [list_[0]] +
                qs([i for i in list_[1:] if i >= list_[0]]))
```

done

counting quick sort: $n = 7$

done (last week)

$$qs([4, 2, 6, 1, 3, 5, 7])$$

$$qs([2, 1, 3]) + [4] + qs([6, 5, 7])$$

$$qs([1])+[2]+qs([3]) \quad + \quad [4] \quad + \quad qs([5])+[6]+qs([7])$$

$$[1] \; + \; [2] \; + \; [3] \quad + \quad [4] \quad + \quad [5] \; + \; [6] \; + \; [7]$$

$$[1, 2, 3] \quad + \quad [4] \quad + \quad [5, 6, 7]$$

$$[1, 2, 3, 4, 5, 6, 7]$$

merge sort

idea: break the list into halves, merge sort the halves, then merge the sorted halves.

we omit the code for merge — see lab #9

```
def ms(L):
    """
    Produce copy of L in non-decreasing order
    """
    if len(L) < 2 :
        return L[:]
    else :
        return merge(ms(L[:len(L) // 2]),
                     ms(L[len(L) // 2 :]))
```

already sorted

counting merge sort, $n = 8$

$\propto \lg n$ splits

ms([4, 2, 6, 8, 1, 3, 5, 7])

mg(ms([4, 2, 6, 8]), ms([1, 3, 5, 7]))

mg(mg(ms([4,2]), ms([6, 8])), mg(ms([1, 3]), ms([5, 7])))

mg(mg(mg(ms[4], ms[2]), mg(ms([6]), ms([8]))), mg(mg(ms([1]), ms([3])),
mg(ms([5]), ms([7]))))

mg(mg(mg([4], [2]), mg([6], [8])), mg(mg([1], [3]), mg([5], [7])))

mg(mg([2, 4], [6, 8]), mg([1, 3], [5, 7]))

mg([2, 4, 6, 8], [1, 3, 5, 7])

$\propto \lg n$ merge with $\propto n$ copies

[1, 2, 3, 4, 5, 6, 7, 8]

$\boxed{n \times \lg n}$

$\mathcal{O}(n)$

The stakes are very high when two algorithms solve the same problem but scale so differently with the size of the problem (we'll call that n). We want to express this scaling in a way that:

done

- is simple
- ignores the differences between different hardware, other processes on computer
- ignores special behaviour for small n

big-O definition

Suppose the number of "steps" (operations that don't depend on n, the input size) can be expressed as $t(n)$. We say that $t \in \mathcal{O}(g)$ if:

> *there are positive constants c and B so that for every natural number n no smaller than B,* $t(n) \leq cg(n)$

use graphing software on:

$$t(n) = 7n^2 \qquad t(n) = n^2 + 396 \qquad t(n) = 3960n + 4000$$

to see that the constant c, and the slower-growing terms don't change the scaling behaviour as n gets large if $t \in \mathcal{O}(n)$, then it's also the case that $t \in \mathcal{O}(n^2)$, and all larger bounds

$$\mathcal{O}(1) \subseteq \mathcal{O}(\lg(n)) \subseteq \mathcal{O}(n) \subseteq \mathcal{O}(n^2) \subseteq \mathcal{O}(n^3) \subseteq \mathcal{O}(2^n) \subseteq \mathcal{O}(n^n) \ldots$$

sequences

```
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 operation for 32-bit ints

all operations constant wrt n

→ $\Theta(1)$

How does the running time of silly depend on n?

loops

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

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

— constant

$i = 0, \ldots, n-1$ — n iterations

— constant

$\Theta(n)$

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

$i = 0, 1, \ldots, n//2 - 1$ — $\propto n$ iterations

$j = 0, 1, \ldots, n^2 - 1$ — $\propto n^2$ iterations

— constant

$\Theta(n^3)$ iterations

more loops

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

not updated inside loop!

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

- constant

$i = 0, 1, \ldots, \sqrt{n}$] $\propto \sqrt{n}$ iterations

$j = 0, 1, \ldots, \sqrt{n}$] $\propto \sqrt{n}$ iterations only for $i = 0$!

constant

$\Theta(\sqrt{n})$

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

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

- constant

$i = 0, 1, \ldots, n^2 - 1$] $\propto n^2$ iterations

constant

$\Theta(n^2)$

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
```

$\Theta(n^2)$

$\Theta(n)$

Worst case

$\Theta(n^2)$

(even n)

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

$n, n//2, n//4, \ldots, 1$] $\propto \lg n$ iterations

$\Theta(\lg n)$

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 $\log_a c = b$.
- 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:

lg

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

lg

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

miscellaneous

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

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

doesn't involve n!

$O(1)$

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)
```

— $n = 0, 1, \ldots, n-1$

— $m = 0, 1, \ldots, m-1$

$\Theta(n \times m)$

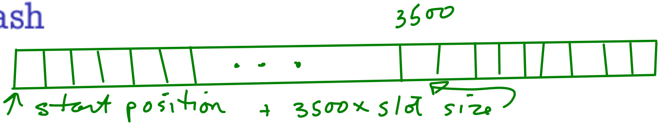
summary

sequences: max

loops: – count iterations, products for nested loops

conditions: – max

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 point in slow-but-clever
- 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.
  if all strings hash to 42, we have a problem...

you can't hash everything!

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

[0, 1]

OR ?

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.

recall

$0 \leq m \% n < n$

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)?

(try it — it took us 17).

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

aka open addressing

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

chaining: keep a small (one hopes) list at that 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 (about 1/3 empty for tim Peters)

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.

double when > 2/3 full

amortized

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