strike still on - guaranteed min funding
if no TAs next week, vote on "status quo"
vs
"some approximation"

- No lab/quiz,
do work handout.

CSC148 winter 2015

efficiency

week 11

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~heap/148/W14/
416-978-5899

March 30, 2015

Outline

searching

height analysis

sorting

big-Oh on paper

big-Oh examples

__contains__

Suppose v refers to a number. How efficient is the following statement in its use of time?

```
v in [97, 36, 48, 73, 156, 947, 56, 236]
```

n - size of list.

Roughly how much longer would the statement take if the list were 2, 4, 8, 16,... times longer? - roughly linear, ~ n

Does it matter whether we used a built-in Python list or our implementation of **LinkedList**? ~ n

add order...

Suppose we know the list is sorted in ascending order, see sorted_list.py

binary search — proportional to lg n
- choose middle element, restrict search.

How does the running time scale up as we make the list 2, 4, 8, 16,... times longer?

$\lg(n)$

Key insight: the number of times I repeatedly divide n in half before I reach 1 is the same as the number of times I double 1 before I reach (or exceed) n: $\log_2(n)$, often known in CS as $\lg n$, since base 2 is our favourite base.

mean base 2

For an n-element list, it takes time proportional to n steps to decide whether the list contains a value, but only time proportional to $\lg(n)$ to do the same thing on an ordered list. What does that mean if n is 1,000,000? What about 1,000,000,000?

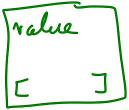
about 50% longer binary search

but 1000 x longer for linear search

trees

How efficient is **__contains__** on each of the following:

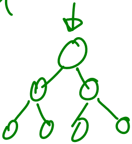
- our general **Tree** class?

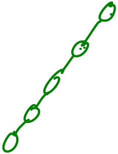


linear
proportional
to n

- our general **BTNode** class?

linear





- our **BST** class?

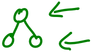
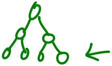
The last case should probably be answered "depends..."

node packing...

maximum number of nodes in a binary binary tree of height:

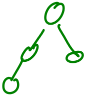
binary

binary

- 0 — 0 | 0
- 1? — 1 | 1
- 2? — 3 | 2 
- 3? — 7 | 4 
- 4? — 15 | 8
- h? — $2^0 + 2^1 + \dots + 2^{h-1} = 2^h - 1$

Complete binary tree: every level full (of nodes) except, possibly, leaf level.

min # nodes in complete BT height



eg

$2^{h-1} \le n \le 2^h - 1 < 2^h$

$h-1 \le \lg(n) < h$

invert node packing...

prev page.

if $n \leq 2^h - 1$, then $n + 1 \leq 2^h$. Take lg from both sides:

$$\lg(n+1) \leq h$$

... where h is the height of the tree

notice that $\lg(n) \equiv \lg(n+1)$, so if our BST is tightly packed (AKA balanced), we use proportional to $\lg(n)$ time to search n nodes

sorting

how does the time to sort a list with n elements vary with n?

it depends:

$\sim n^2$

- bubble sort each pass swaps out-of-order elements linear
  $\sim n$ ops/pass
  $\sim n$ passes.
- selection sort find value for each pos.
  min next min & so
  $\underbrace{n + n-1 + n-2 + \ldots + 2,}$
- insertion sort $1 + 2 + 3 + \ldots + n$ $\frac{n(n+1)}{2}$
  for each value, find ith position
  $\frac{n^2+n}{2}$
  $\underbrace{1 + 2 + 3 + \ldots + n-1}$
  $\frac{n^2+n}{2}$
- some other sort?

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(L):
    ''' (list) -> list
    '''
    if len(L) < 2:
        # copy of L
        return L[:]
    else:
                                element < L[0]
        return (qs([i for i in L if i < L[0]]) +
                [L[0]] +
                qs([i for i in L[1:] if i >= L[0]]))
                        ≥ L[0], but not L[0] itself
```

counting quick sort: $n = 7$

quick_sort : $n \lg n$

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

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

n n steps

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

[1] + [2] + [3] + [4] + [5] + [6] + [7]

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

n n

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

$\mathcal{O}(n)$

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:

- 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!')
```

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
```

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
```