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

Computer Science
UNIVERSITY OF TORONTO

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]
```

Roughly how much longer would the statement take if the list were 2, 4, 8, 16,... times longer?
Does it matter whether we used a built-in Python list or our implementation of **LinkedList**?

add order...

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

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.

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?

trees

How efficient is _**contains**_ on each of the following:

- our general **Tree** class?
- our general **BTNode** class?
- our **BST** class?

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

node packing...

maximum number of nodes in a binary tree of height:

- 0
- 1?
- 2?
- 3?
- 4?
- n?

invert node packing...

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:

- bubble sort
- selection sort
- insertion sort
- 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:
        return (qs([i for i in L if i < L[0]]) +
                [L[0]] +
                qs([i for i in L[1:] if i >= L[0]]))
```

counting quick sort: $n = 7$

$$\text{qs}([4,\ 2,\ 6,\ 1,\ 3,\ 5,\ 7])$$

$$\text{qs}([2,\ 1,\ 3]) + [4] + \text{qs}([6,\ 5,\ 7])$$

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

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

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

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

$\mathcal{O}(n)$

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
```