CSC148 winter 2018

efficiency considerations

week 10

Danny Heap

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

http://www.teach.cs.toronto.edu/~csc148h/winter/

416-978-5899

March 19, 2018

Outline

searching

height analysis

sorting

big-Oh on paper

big-Oh,Omega,Theta 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?

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

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?
- h?

invert node packing...

if $n < 2^h \leq 2n$, then take lg from both sides:

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

...where h is the minimum height of the tree to pack n nodes

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(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]]))
```

counting quick sort: $n = 7$

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

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

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

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

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

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

merge...

```
def merge(L1, L2):
    """return merge of L1 and L2
    """
    L = []
    i1, i2 = 0, 0
    while i1 < len(L1) and i2 < len(L2):
        if L1[i1] < L2[i2]:
            L.append(L1[i1])
            i1 += 1
        else:
            L.append(L2[i2])
            i2 += 1
    return L + L1[i1:] + L2[i2:]
```

merge sort

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

$\mathcal{O}(t), \Omega(t), \Theta(t)$

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
```

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
```

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
```

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
```

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:

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

- 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
```

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)
```

summary

sequences:

loops:

conditions: