CSC148 L5102
Introduction to Computer Science
Joe Lim
joe.lim@utoronto.ca
Office Hours: 5-6pm PB150
http://www.cdf.toronto.edu/~csc148h/winter/

Outline
- Recursion Efficiency
- Searching
- Height Analysis
- Sorting
- Big-Oh on paper

Reminders
- A2 Due Date: March 24 @ 10:00p.m.

Redundancy
Some recursive functions "write themselves" - you write down the base case and general case from a definition, and you have a program

```python
def fibonacci(n):
    """Return the nth fibonacci number, that is, n if n < 2, or fibonacci(n-2) + fibonacci(n-1) otherwise."
    @param int n: a non-negative integer
    @type: int
    """  pass
```

Expand...
Break our usual rule about expanding a branching recursive in order to see how much computation is spawned by fibonacci(29)

```python
if n < 2:
    return n
else:
    return fibonacci(n-2) + fibonacci(n-1)
```

Solution? Memoize
```python
def fib_memo(n, seen):
    """Return the nth fibonacci number reasonably quickly."
    @param int n: index of a fibonacci number
    @param dict[int, int] seen: already-seen results
    @type: int
    """
    if n not in seen:
        seen[n] = (n if n < 2 else fib_memo(n-2, seen) + fib_memo(n-1, seen))
    return seen[n]
```
Running out of stack space

Some programming languages have better support for recursion than others; python may run out of space on its stack for recursive function calls ...

Sometimes you can re-set system defaults (see A2’s puzzle_tools.py)

___contains___

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

```python
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 \( \log(n) \) to do the same thing on an ordered list.

What does that mean if \( n = 1,000,000 \)? What about 1,000,000,000?

Trees

How efficient is ___contains___ on each of the following

- Our general Tree class?
- Our general BTNNode 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 $n < 2^h \leq 2n$, 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 re-combine the list:

```python
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: $n = 7$

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

$O(n)$

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:

- 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 $t \in O(g)$ 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:

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

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

\[
O(1) \subseteq O(\log(n)) \subseteq O(n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n) \subseteq O(3^n)...
\]