CSC148 winter 2017

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 24, 2017
Outline

recursion efficiency

searching

height analysis

sorting

big-Oh on paper
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
    @rtype: int
    """
    pass
```
break our usual rule about expanding a branching recursive in order to see how much computation is spawned by \texttt{fibonacci(29)}

\begin{verbatim}
if n < 2:
    return n
else:
    return fibonacci(n-2) + fibonacci(n-1)
\end{verbatim}
def fib_memo(n, seen):
    """
    Return the nth fibonacci number reasonably quickly.
    """
    @param int n: index of fibonacci number
    @param dict[int, int] seen: already-seen results
    """
    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]
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 puzzle_tools.py)
Suppose \( v \) refers to a number. How efficient is the following statement in its use of time?

\[
v \text{ in } [97, 36, 48, 73, 156, 947, 56, 236]
\]

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

\[ h \leq \log(n) + 1 \]

... 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 $\log(n)$ time to search $n$ nodes
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: $n = 7$

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

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


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

$$[1, 2, 3, 4, 5, 6, 7]$$
\( \mathcal{O}(t), \Omega(t), \Theta(t) \)

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 $t \in O(g)$ 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

$O(1) \subseteq O(\log(n)) \subseteq O(n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n) \subseteq O(n^n) \ldots$