Time efficiency and big-oh

csc148, Introduction to Computer Science
Diane Horton
Winter 2015

UNIVERSITY OF TORONTO

DCS50

Ways to store a sequence of objects

- Suppose we want to store a sequence of ints.
- What are some structures we could use?
  - Python list
  - linked list
  - binary tree
  - binary search tree
  - tree with larger branching factor

- How do we choose one?
- Speed is one factor.

Speed of search in a Python list

- Suppose v refers to some int.
- How much work is required to do this:
  `v in [97, 36, 48, 73, 156, 947, 56, 236]`
- How much longer would it take if the list were
  - 100 times longer?
  - 1000,000,000 times longer?
- What would our answers be if we used our implementation of LinkedList instead?

Speed of search in a Python list

- With either a Python list or a linked list of n elements:
  - we must look at elements one by one.
  - each time we eliminate only one element from consideration.
  - in the worst case we look at all n elements.
- Either way, it takes time proportional to n.

- Can we make search in a Python list faster?

Speed of search in a *sorted* Python list

- Now we can do binary search instead:

```
def contains(L, v):
    if len(L) == 0:
        return False
    elif len(L) == 1:
        return L[0] == v
    else:
        mid = len(L) // 2
        if L[mid] <= v:
            return contains(L[mid:], v)
        else:
            return contains(L[:mid], v)
```

- How much work is search now?

Speed of search in a *sorted* Python list

- How many times can we split a list in half before we are down to one element?

- list of length 1: 0 times
- list of length 2:
- list of length 4:
- list of length 8:
- list of length 16:

Speed of search in a *sorted* Python list

- In general, we can split a list of length n in half $\log_2 n$ times before its size is 1.
- So searching a sorted list of n elements, takes time proportional to lg(n).

- Can we get the same speed-up for sorted linked lists?

Aside: logarithms

- $\log_b y = x \iff b^x = y$
- Example:
  $4^3 = 64 \iff \log_4 64 = 3$
- lg is binary logarithm. A base of 2 is implied.
- Example:
  $\lg(32) = \log_2 32.$

What about search in a tree?

- How efficient is search in
  - our general Tree class?
  - binary trees, as defined in our BTNode class?
  - binary search trees, as defined in our BST class?

Speed of search in a tree

- Same as with a Python list or a linked list of n elements, in a tree:
  - we must look at elements one by one.
  - each time we eliminate only one element from consideration.
  - in the worst case we look at all n elements.
- Either way, it takes time proportional to n.

- Can we make search in a tree faster?

Speed of search in a binary *search* tree

- Now we can exploit the tree's ordering to go left or right at each step.
- So we potentially eliminate not just one element, but half the yet-to-be-considered parts of the tree.

Number of nodes in terms of height

height	maximum number of nodes in a binary tree
1	1
2	3
3	7
4	15
5	31
h	$2^h - 1$

Height in terms of number of nodes

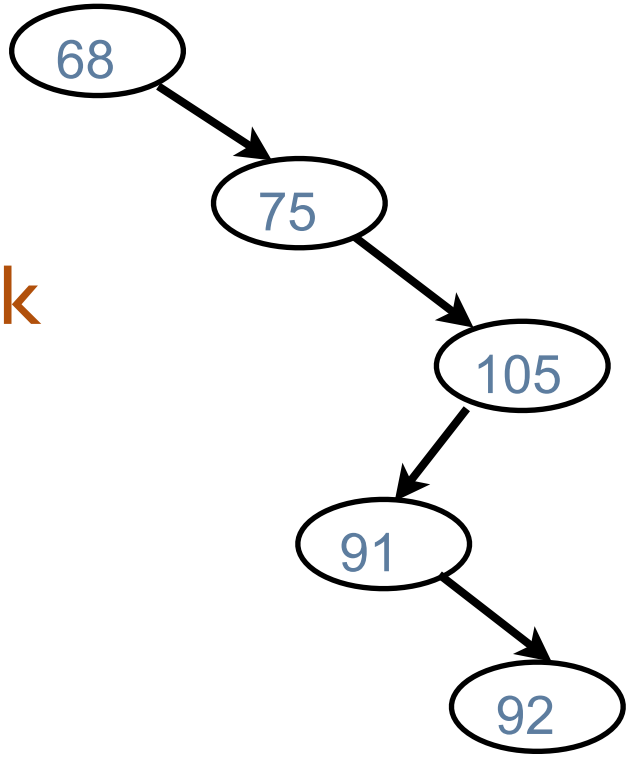
- Let's flip that around to figure out tree height for a fixed number of nodes.
- Let n be the number of nodes in a tree of height h.
- We showed that $n \leq 2^h - 1$.
- $n + 1 \leq 2^h$.
- $\log_2 (n + 1) \leq h$.
- $h \geq \log_2 (n + 1)$.

Speed of search in a binary *search* tree

- Search in a BST requires only traversing one branch.
- In the worst case, we go all the way to a leaf.
- So search in a BST with n nodes uses time proportion to lg(n).
- Or does it?

Speed of search in a binary *search* tree

- … Only if the tree is "balanced"!
- E.g., this tree takes time proportional to n to search:
- In csc263, you'll learn about balanced trees such as red-black trees or AVL trees.



When a balanced BST isn't fast enough

- Databases have to deal with so much data that it can't fit in memory.
  - So it's stored in a tree that's in a file
- References become locations in a file rather than locations in memory.
  - Following a reference in a file is *much* slower.
- So we must have very short trees.
  - Solution: huge branching factors.
- In csc343 and csc443, you'll learn about databases and their implementation.

Speed of sorting: mergesort

- Consider the tree of calls to mergesort.
- How many levels are in the tree?
  - There are $\log_2 n$ levels.
- How much total work is involved to do all the merging required at each level?
  - It is proportional to n.
- So to mergesort a list of n items requires time proportional to $n \times \log_2 n$.

Speed of sorting: other sorts

- Many familiar sorts are much slower: they require time proportional to n^2.
  - Bubble sort, selection sort, insertion sort.
- Other sorts require time proportional to n x $\log_2$ n, like mergesort.
  - quicksort, heap sort, shell sort.
- This is only part of the story.
  - E.g., we also care about *average* case.
  - You'll learn more about this in csc263.

Time analysis of recursive code

- Our analysis of mergesort was quite informal.
- It also required unwind all the recursive calls.
- When you understand recursion well, you don't have to unwind in order to write it.
- You don't have to unwind it to analyze it either.
- In csc236, you'll learn how to use recurrence relations to analyze the time efficiency of recursive code.

Big vs small differences

- Let's return our attention to iterative code.
- Search in a general tree vs a binary tree?
  - About the same: proportional to n.
  - Any differences in speed are small-ish.
- How about search in a binary search tree?
  - Much faster: proportional to $\log_2 n$.
  - This is a big difference.
- Before we fine-tune, we should pick the best!
  - Fine-tuning search in a binary tree will never make it as fast as in a BST!

How do we know which is faster?

- We can code them up and race them.
  - As you are doing in the labs.
- But we don't have to.
  - We can compare the *algorithms*.
- Then we can implement the best one.
  - Saves much effort!

Is an algorithm even feasible?

- Minimax considers all game states.
  - How many game states are there for 3x3 tippy?
  - How many for 4x4 tippy?
- Considering all groupings of n students into groups of size ~k.
  - Even for n=22, it is infeasible to consider all possible groupings.
- In csc463 you will learn about "feasible".
- If our best algo isn't feasible, should we keep trying to find a better one?

Is a problem itself feasible?

- Two problems:
  - Find the shortest path through a network that hits every node once and returns to the start?
  - Find the "cheapest" way to connect every node in a network.
- Seem similar, but one is feasible and one is not.
- Sometimes we can show that *no* algorithm can solve a problem in reasonable time.
- In csc373, you'll learn tactics for dealing with that.

Formalizing: definition of big-O

- Suppose the number of "steps" (operations that don't depend on the input size) can be expressed as t(n).
- We say that $t(n) \in O(g)$ if

  there are positive constants c and B such that for every natural number n no smaller than B,
  $t(n) \leq c \times g(n)$.

Properties of big-O

- If $t \in O(n)$, then $t \in O(n^2)$ also, and all the larger bounds.

- $O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq O(2^n) \subseteq O(n^n)$

Using big-oh to analyse code

- Let's analyze a bunch of code snippets using big-oh. [code.py]

Summary

Type of code	Technique
doesn't depend on size of inputs	O(1)
sequence	add
loop	multiply
branching	take the maximum