Week 10: Efficiency

Arnamoy Bhattacharyya

Efficiency

1. More empirically than CSC165
2. Talk mainly about time complexity
    a. Using timing tools available to us
        i. E.g datetime() in Python
        ii. gprof for c/c++
        iii. Unix time command

Recap: Fibonacci Recursive

- Remember recursion:
- Calculating Fibonacci numbers • if n < 2, fib(n) = 1 • fib(n) = fib(n-1) + fib(n-2)
- Write a recursive program for this..

```
def fib(n):
    """
    Returns the n-th fibonacci number.
    @param int n: a non-negative number
    @rtype: int
    """
    pass
```

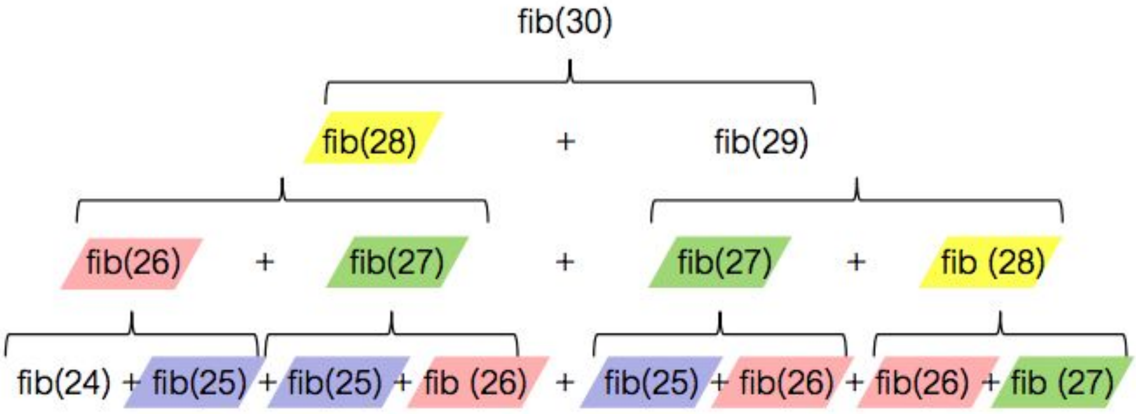
Recap: Fibonacci Recursive

- Remember recursion:
- Calculating Fibonacci numbers • if n < 2, fib(n) = 1 • fib(n) = fib(n-1) + fib(n-2)
- Write a recursive program for this..

```
def fib(n):
    """
     Returns the n-th fibonacci number.
     @param int n: a non-negative number
     @rtype: int
    """
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Redundancy

• Unnecessary repeated calculations => inefficient! • Let's expand the recursion: fib(n) = fib(n-1) + fib(n-2)



How could we avoid calculating items we already calculated?

Solution? Memoize

- Keep track of already calculated values

```
def fib_memo(n, seen):
    """
     Returns the n-th fibonacci number, reasonably quickly, without redundancy.
     @param int n: a non-negative number
     @param dict[int, int] seen: already-seen results
     @rtype: 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]
```

One more example of memoize

```
def count_states (s1: SubtractSquareStates) ->int:
    moves = s1.get_possible_moves()
    states = [s1.make_move(m) for m in moves]
    return 1 + sum([count_states(x) for x in states])
```

One more example of memoize

```
def count_states (s1: SubtractSquareStates,
                  seen:dict) ->int:
    if s1.__repr__() not in seen:
        moves = s1.get_possible_moves()
        states = [s1.make_move(m) for m in moves]
        seen[s1.__repr__()] = 1 + sum([count_states(x) for x
in states])
    return seen[s1.__repr__()]
```

Efficiency Considerations

- How you implement matters
- You can code up fast a really inefficient code
- If you think about efficiency, you will be gem
- Key is to identify *which* parts are inefficient
    - How the ***time*** grows with ***input***
    - e.g fib(input), GameState(input)

Recursive vs iterative

- Any recursive function can be written iteratively
- May need to use a recursive data_structure too, potentially
- Recursive functions are not more efficient than the iterative equivalent
- Why ever use recursion then?
- If the nature of the problem is recursive, writing it *iteratively* can be
  - a) more time consuming, and/or
  - b) less readable

Recursive functions are not more efficient than their iterative equivalent
But .. Recursion is a powerful technique for naturally recursive problems

Efficiency considerations: Search speed

contains in a list

- Suppose v refers to a number:

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

- What is an example of worst case value for v?
  - In terms of number of nodes compared?

__contains__

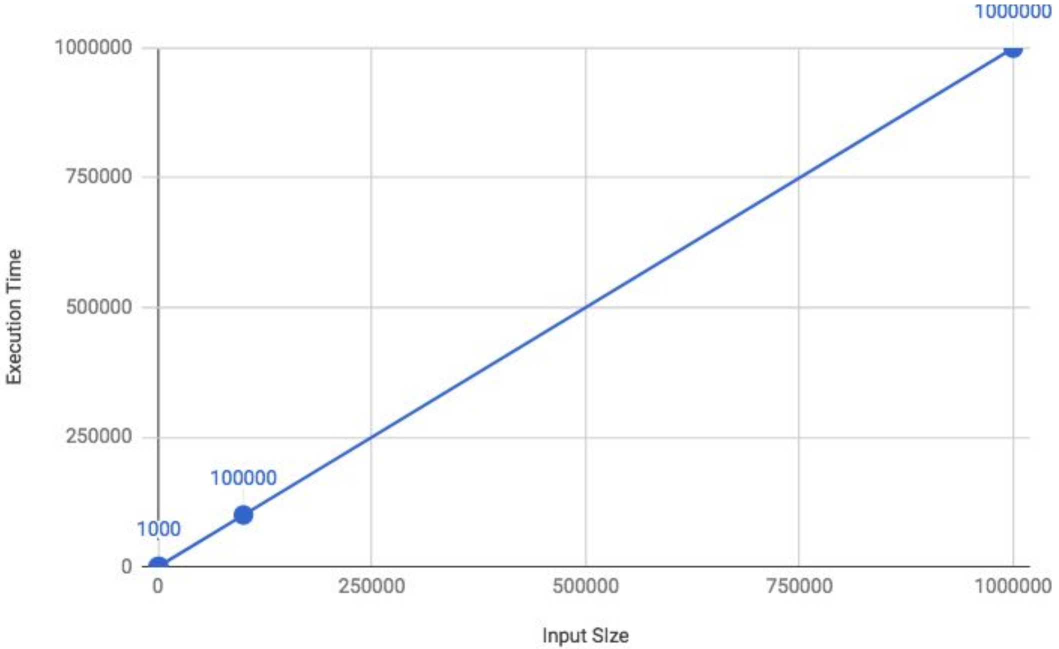
v not in list performance:

10 items → 10 comparisons

1000 items → 1000 comparisons

1000000 items → 1000000 comparisons

__contains__



Linear Growth

O(n)

__contains__

Suppose v refers to a number:

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

how to improve?

__contains__ how to improve?

Suppose v refers to a number:

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

Sort it

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

E.g 170 in list → 3 comparisons compared to 8

__contains__ on sorted list

How many times can you keep halving a value until you reach 1:

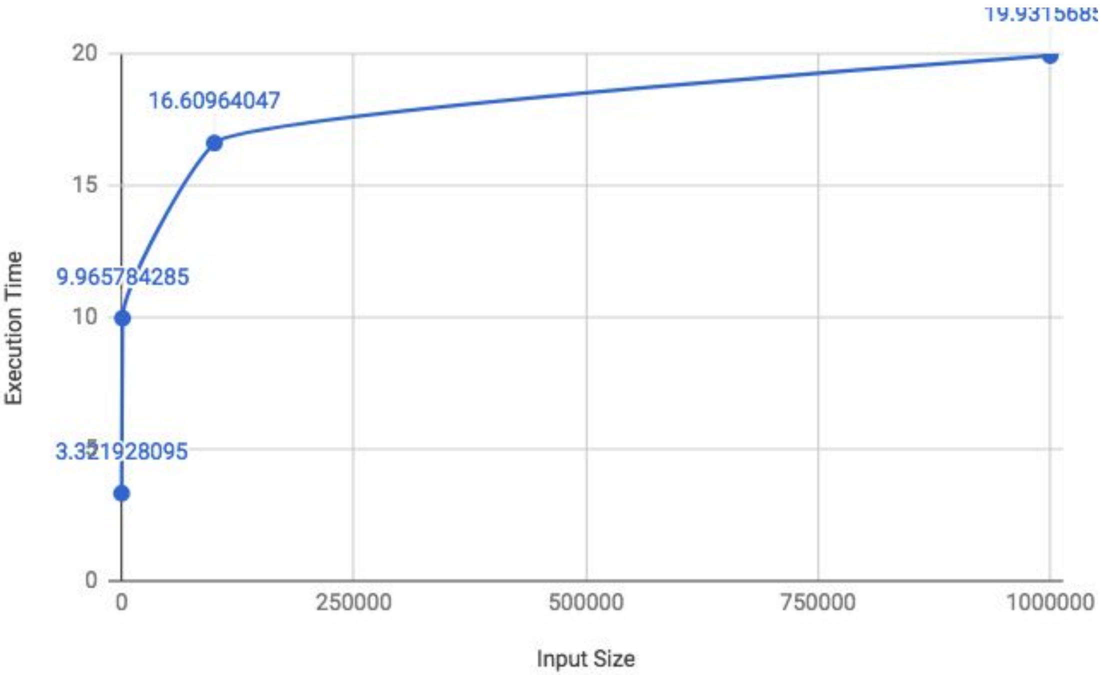
8 → 3 times

16 → 4 times

32 → 5 times

n → ?

__contains__ on sorted list



Logarithmic Growth

$O(\log_2 n)$

or

$O(\lg n)$

__contains__ basic vs optimized

Input Size	**__contains__**	**__contains__ sorted**
10	10	3
1000	1000	10
100000	100000	17
1000000	1000000	20

Efficiency in trees, contains()

What is the worst case while finding a value in a tree?

Efficiency in trees, contains()

What is the worst case while finding a value in a tree?

Execution time

General Tree?

Binary Tree?

BST?

contains in BST: Height of a tree

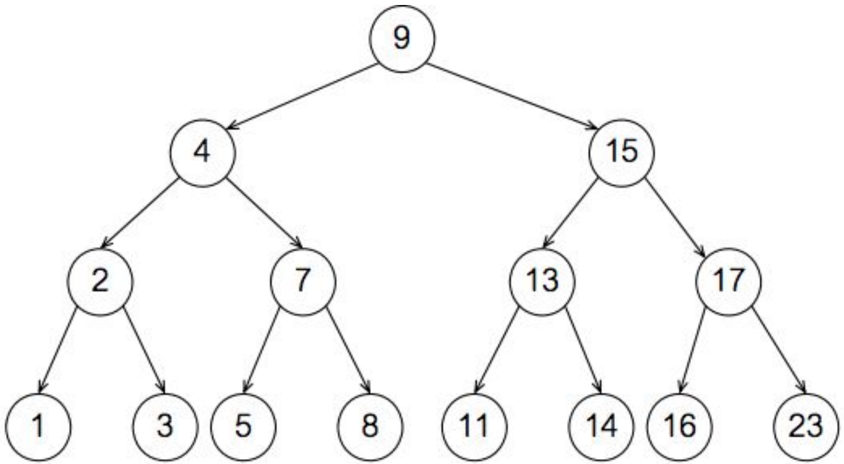
- We know that $n \leq 2^h - 1$

  $\Rightarrow n + 1 \leq 2^h$

  $\Rightarrow \log_2 (n + 1) \leq h$

  $\Rightarrow h \geq \log_2 (n + 1)$

- So, time will be proportional to *lg n*



Exercise

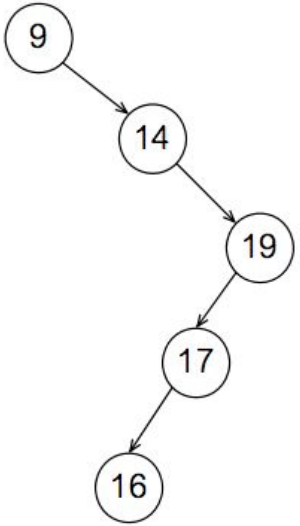
When will BST search not be O(lg n)?

BST search NOT ALWAYS log(n)

Imbalanced Tree -- O(n)

You will learn more about self-balancing trees

Later (AVL trees, Red-Black trees)



Quicksort

Idea:

Choose an item as *pivot*

Put items < pivot on the left

Put items > pivot to the right

Keep recursing on left and right

| **12** | 4 | 3 | 9 | 43 | 16 | 56 | 1 |

| 4 | 3 | 9 | 1 | **12** | 43 | 16 | 56 |

| **4** | 3 | 9 | 1 | **12** | **43** | 16 | 56 |

Quicksort

```python
def qs(list_):
    """
     Return a new list consisting of the elements of list_ in ascending order.
     @param list list_: a list of comparables
     @rtype: list
    """
    if len(list_) < 2:
        return list_[:]
    else:
        smaller = [i for i in list_[1:] if i < list_[0]]
        larger = [i for i in list_[1:] if i >= list_[0]]
        return (qs(smaller) +
                [list_[0]] +
                qs(larger) )
```

Lists of length < 2 are already sorted

Simpler partition step

Sort smaller elements

in its correct position

Sort larger elements

Counting quicksort: n = 7

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

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

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

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

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

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

lg(n) splits

n comparisons at each split

lg(n) concatenation, with n work to concatenate

n x lg n

Worst case of quick sort

List already sorted

What is the big-oh performance? Is it still n * lg(n)?

Performance of other sorting algorithms

- bubble sort -> n^2
- selection sort -> n^2
- insertion sort -> n^2