COMP_SCI 214: Data Structures and Algorithms

Complexity and Big-O

PROF. SRUTI BHAGAVATULA

Announcements

- Homework 1 feedback is out
  - Got something wrong? Don’t panic; resubmit!
    - Report tells you what went wrong
  - Larger mistakes? You still have things to learn!
    - And you should get credit for learning them!
- Homework 1 self-eval is out (due Thursday)
  - Goal: evaluate code beyond functional correctness
  - HW handout told you what to pay attention to
  - Read directions in eval carefully
  - <u>**Only based on your first submission**</u> (download original submission if needed)

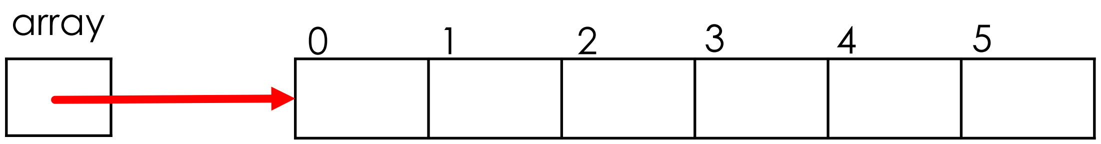
Homework 2 advice

- Use SLL, stack array, ring buffer, and dynamic array for data representation inspiration
- Draw data representation and operations out **before** programming
  - When debugging, draw out what your program is doing and ensure invariants are satisfied!
- Watch "Supplementary videos", especially
  - "Common errors in DSSL2": aliasing (`#0#`) and other things
  - "Classes, objects, and interfaces in DSSL2"

Comparing data representations and programs

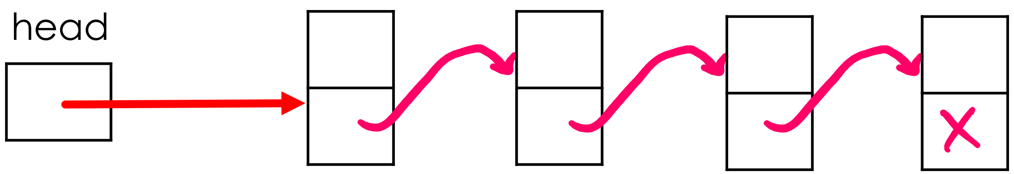
Array vs. Linked list

- **Getting value at position 5 in array. (`[i]` operation)**



- How long does it take to run?
  - 0.00081 seconds

- **Getting value at position 3 in linked list (`get_ith()` function)**



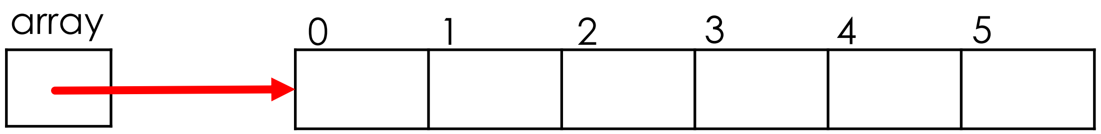
- How long does it take to run?
  - 0.0034 seconds

Are these the answers we want?

What if I just ran the array operation on a faster computer?

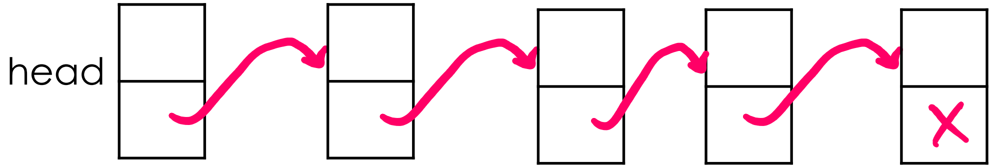
Array vs. Linked list

- Get value at position `pos` in array



- Steps:
  1. `array[position]`

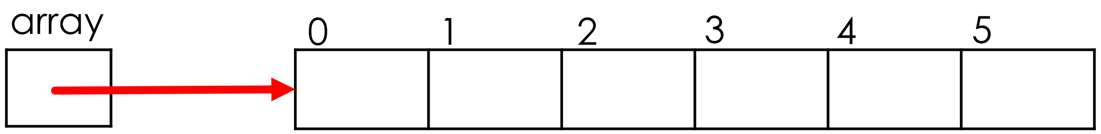
- Get value at position `pos` in linked list



- Steps:
  1. `count = 0`
  2. `curr = head`
  3. `while curr is not None:`
     a. `if count == position:`
        i. `return curr.data`
     b. `count = count + 1`
     c. `curr = curr.next`

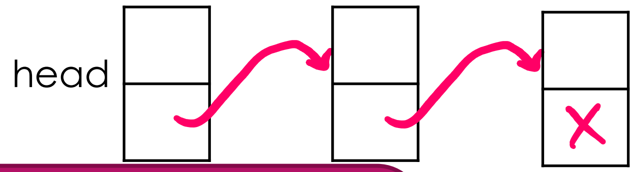
Array vs. Linked list

- Get value at position `pos` in array



- Steps:
  1. `array[pos`

- Get value at position `pos` in linked list



Which seems like it would take longer?

```
not None:
    a. if count == position:
        i. return curr.data
    b. count = count + 1
    c. curr = curr.next
```

What do we want in our answer to "how long"?

- General:
  - Applicable to a large class of programs and algorithms
  - Independent of particular hardware
  - Applicable to many types of resources (time, space, etc.)
- Useful:
  - Don't need to predict actual time
  - Helps us select between various algorithms
- How long? ~ # steps

Getting value in array: number of steps

- Steps:
  1. `array[position] = new_val`
- Total #steps:
  - Recall: array indexing is the same speed regardless of index
- Concrete examples:
  - `[idx]` on array with `n = 5`
    - # Steps: 1
  - `[idx]` on array with `n = 100`
    - # Steps: 1

Getting value at `idx` in LL of size n: Number of steps

- Steps

```
count = 0
curr = head
while curr is not None:
    if count == position:
        return curr.data
    count = count + 1
    curr = curr.next
```

1

1

3

2

1

1

1

1

$n \times 4$

1

- Total #steps (assuming `idx` is at the end): $2 + 4n + 1$

Getting value in linked list: number of steps

- Steps:

```
count = 0
curr = head
while curr is not None:
    if count == position:
        return curr.data
    count = count + 1
    curr = curr.next
```

- Total #steps (assuming `idx` is at the end): $4n+3$

- Concrete examples:
  - `get_ith` on LL with `n = 5`
    - Steps: 23
  - `get_ith` on LL with `n = 100`
    - Steps: 403

What do steps represent?

- Time an operation takes
- Cost of an operation
- Whether and how time is related to DS size
  - E.g., If steps change with n or not

Complexity

`[]` and `get_ith()`

- Say $f(n) = 1$ (# steps for [] operation)
- Say $g(n) = 4n + 3.$ (# steps for `get_ith()` function)

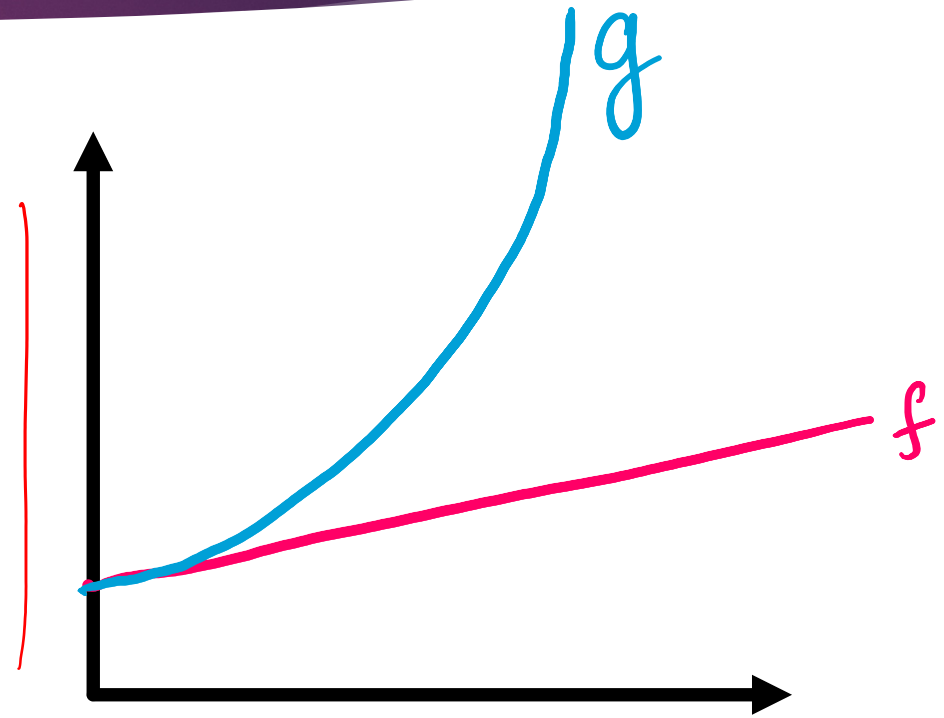
- Which operation would you say is less expensive?
  - $f(n)$ is less expensive

- Formally: $f(n) \leq g(n)$ for all n

More steps function comparisons

- $f(n) = 3n + 3$
- $g(n) = 3n^2 + 3$

- Which is less expensive?
  - $f(n) \leq g(n)$ for all n



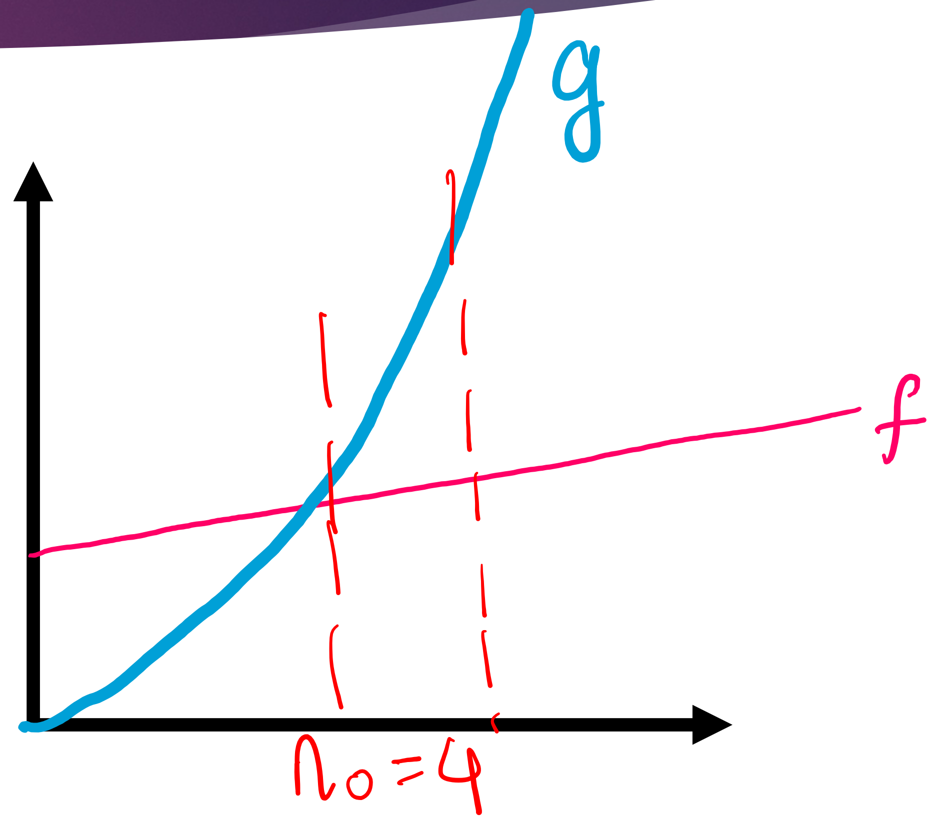
More steps function comparisons

- $f(n) = 3n + 3$
- $g(n) = n^2$

- Which is less expensive?
  - $f(n) \leq g(n)$ for all $n \geq n_0$
  - f(n) is less expensive

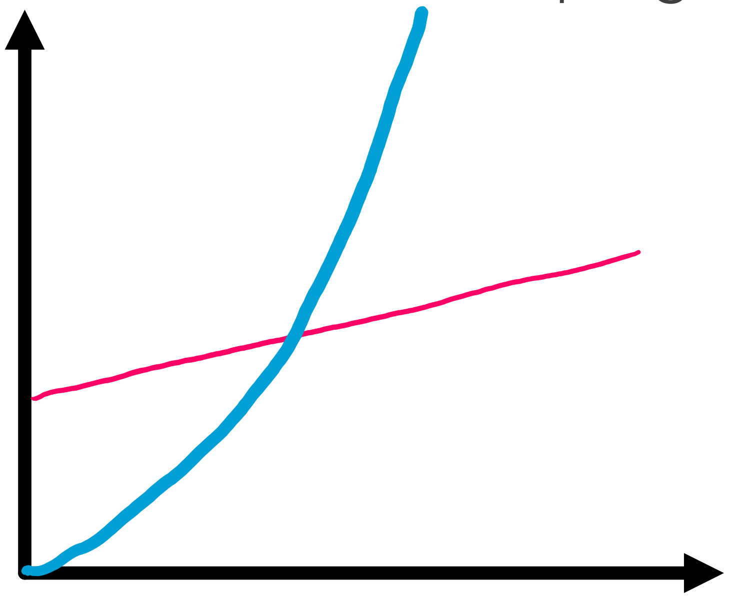
$3 \times 4 + 3$ 16

$15 \leq 16$



Asymptotic cost

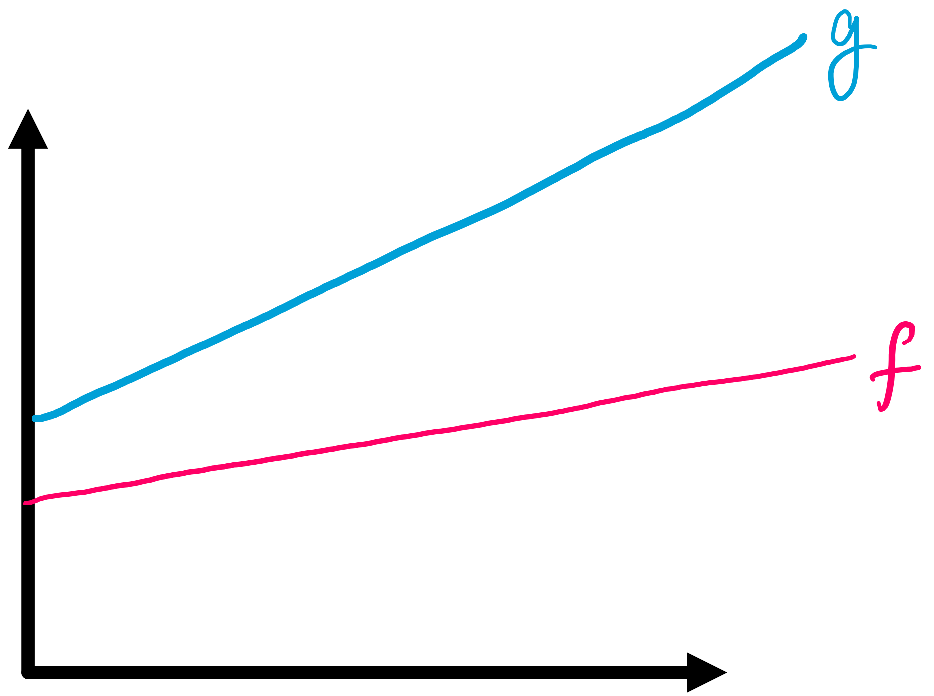
- Comparison only matters when n gets large enough
  - **Asymptotic cost** is cost as n approaches ∞
  - Cost doesn't mean much for a small values of n since programs will be "fast" regardless



Comparing more functions

- $f(n) = 3n + 3$
- $g(n) = 4n + 4$

- Which is less expensive?
  - Seems like $f(n)$ is less expensive by our definition
  - But both are linear functions
  - So is the difference meaningful?

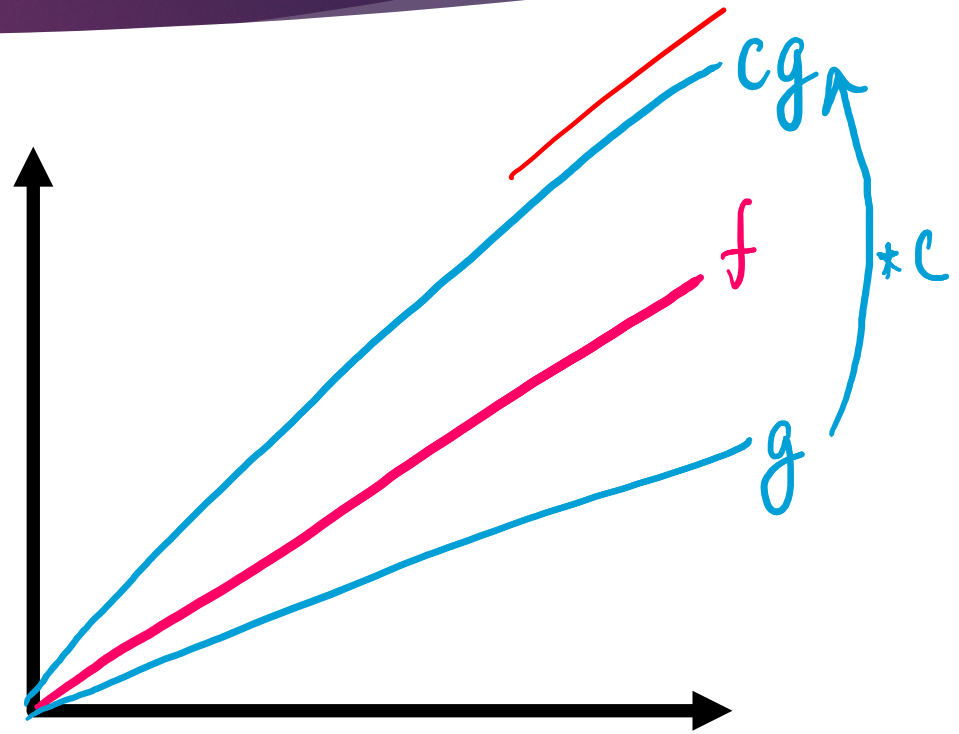


Final definition for comparing costs

- Costs can be "equal" too
  - E.g., both linear, both quadratic

- $f(n) \leq c * g(n)$ for all $n \geq n_0$
  - Implies $f(n)$ is as expensive or less expensive than $g(n)$

- $f(n) \in O(g(n))$



Big-O

- $\mathrm{f(n)} \leq \mathrm{c} * \mathrm{g(n)}$ for all $\mathrm{n} \geq \mathrm{n_0}$
- $\mathrm{f(n)} \in \mathrm{O(g(n))}$
- $\mathrm{O(g(n))} = \{\mathrm{f(n)}: \text{where above definition is satisfied}\}$

- Big-O is a set of functions that contains, in the worst case:
  - Less expensive functions
  - Equally expensive functions (according to inequality above, can differ by constant c)
- "Given a function f, $O(f)$ is the set of functions that 'grow no faster than' f"

Big-O bound

- We often call a Big-O bound:
  - Complexity of a program (“Asymptotic complexity”)
  - Run time of a program
  - Cost of a program

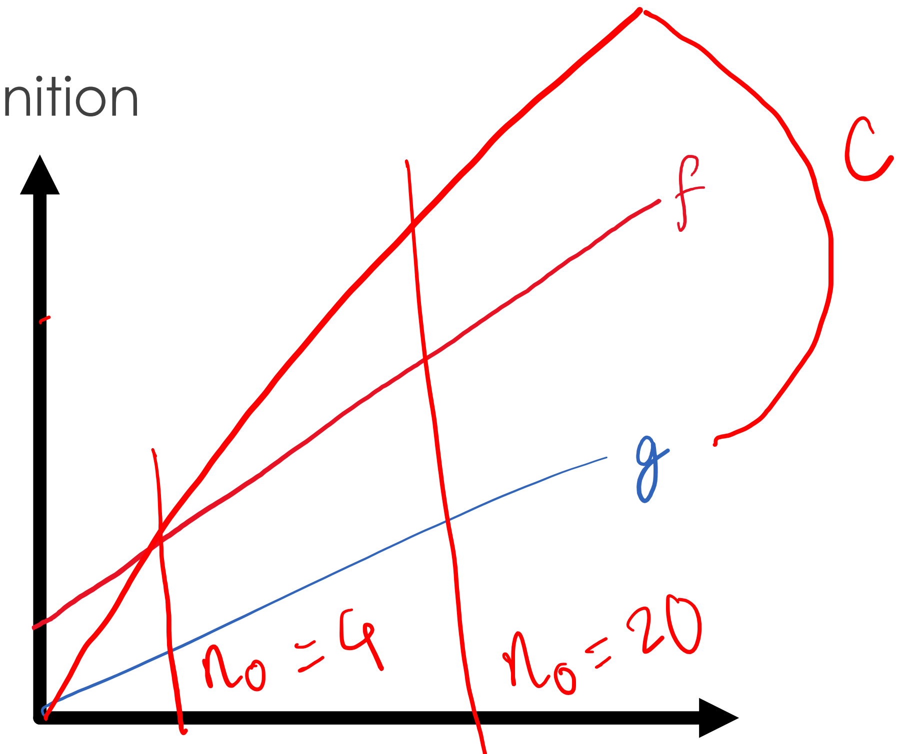
Another Big-O example

- $3n + 3 \in O(n)$ g(n)
  - How so?
  - Let's visualize the definition

$f(n) \leq c * g(n)$ for all $n \geq n_0$

$n_0 = ?$ 4

$c = ?$ 4



- $O(3n + 3) = O(n)$
  - Tightest bound

$3n_0 + 3 \leq c * n_0$

$3 \times 4 + 3 \leq c * n_0$

$15 \leq c * n_0$

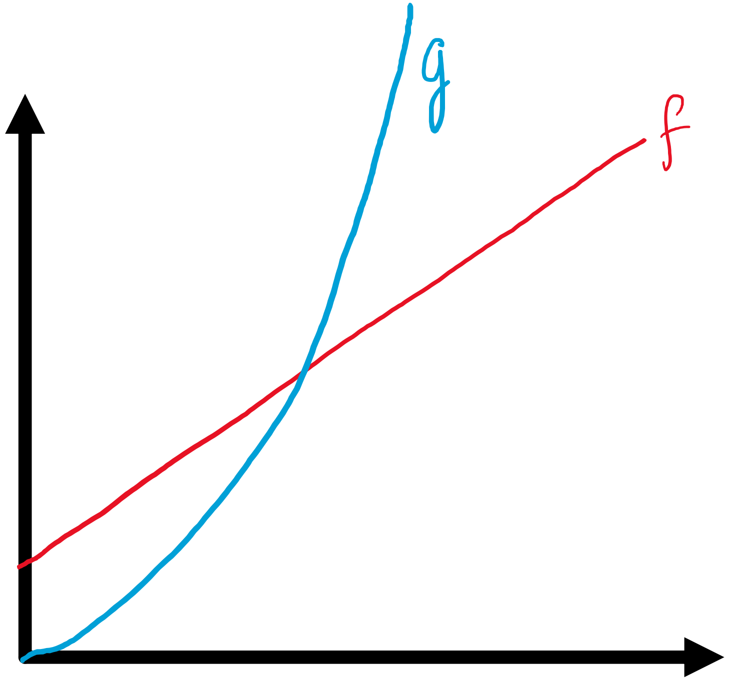
$15 \leq c * 4$

$c = 4$

Another Big-O example

- $3n + 3 \in O(n^2)$
  - How so?
  - From previous slide, $3n + 3 \in O(n)$
    - $O(n) \subseteq O(n^2)$
    - So $3n + 3 \in O(n^2)$
  - But $O(n)$ is the "tightest" bound

$$f(n) \le c * g(n) \text{ for all } n \ge n_0$$



Always use the tightest Big-O bound

- Say a function belongs to $O(n)$ and $O(n^2)$
  - We say it belongs to $O(n)$ which is the smaller set of functions (tight)
  - $O(n^2)$ would not be tight
- Say a function belongs to $O(n)$ and $O(5n + 8)$
  - We say it belongs to $O(n)$ which is the simpler formula (no coefficients and extra constants)
- Tight bound ignores most of the singular or standalone "unimportant" statements; focuses on the type of function growth

How to get the tightest bound

- $f(x) = 3n^2 + \cancel{3n} + \cancel{\frac{1}{2}}$
- $f(x) \in$ $O(n^2)$

1. Ignore constant co-efficients and constant terms
2. Keep dominant term (with highest power)

- Try: $f(x) = 5$ 1
- $f(x) \in$ $O(1)$

To summarize

- A program's cost is roughly described by the number of steps (not time units)
  - Cost can be independent of DS size
  - Cost can change based on DS size
  - Number of steps is reduced to count the most prominent and costly steps
- Cost is described in worst-case scenario using Big-O notation
- Informally, we say a program has complexity O(…) to say that the cost of the program scales at the rate of the function "…"
- Tight bound usually contains highest-order term and no constants (coefficients or terms)

Pause

- Any questions?
- Anything unclear?

Big-O exercises

1. An algorithm looks for duplicate names in an array with this approach: iterate through the array of names; for each name, iterate through all the other names in the array and see if there's a match. What is its tightest Big-O complexity?

```
for i1, name1 in names:
    for i2, name2 in names:
        if name1 == name2 and i1 != i2:
            do something…
```

n

n

$O(1)$

$n \times n \times 1$

$n^2 \in O(n^2)$

Big-O exercises

2. A function iterates through the elements of an array that are at even indices. What is its tightest Big-O complexity?

```
index = 0
while index < arr.len():
    do something…
    index = index + 2
```

$O(1)$

$\frac{n}{2}$

$O(1)$

$\cancel{1} + \left(\frac{n}{2} \cancel{\times 1}\right) = \cancel{\tfrac{1}{2}}\, n \in O(n)$

Big-O exercises

3. Factorial of a number:

$$f(n) = n * (n-1) * (n-2) \ldots * 2 * 1$$

What is its tightest Big-O complexity?

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n-1)
```

O(1)

n = 5
n = 4
n = 3
n = 2
n = 1

1 step
1 step
1 step
1 step
1 step

= 5 ≤ n
∈ O(n)

Takeaways

- The "important steps" that form a tight Big-O bound usually represent:
  - The number of iterations (as a function of input size(s))
  - The number of recursive calls made before a base case is hit (as a function of input size(s))
  - Some combination of the above

- What about if a program contains two sequential loops, or recursive calls in which a loop is executed?
  - Count or do the math! Then reduce to tight bound

Common Big-O complexity classes

Complexity class	Common name for algorithms in class
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	"n log n"
$O(n^2)$	quadratic
$O(2^n)$	exponential

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

A lot of algorithms will fall in one of these classes

Big-O equalities

- $O(f(n) + c) = O\big(f(n)\big)$
  - Equivalently: $f(n) + c \in O(f(n))$
  - **Adding constants doesn't matter**
- $O\big(c * f(n)\big) = O(f(n))$
  - **Multiplying by constants doesn't matter**

Big-O equalities continued

- $O\left(\log_k f(n)\right) = O\left(\log_j f(n)\right)$
  - **Log bases don't matter**
- $O(f(n) + g(n)) = O\left(f(n)\right)$ if $g \ll f$
  - **Adding smaller things doesn't matter**

Big-O beyond time

- Big-O notation can apply to all resources
  - Which was our goal

- Space complexity:
  - How does the amount of space needed scale with the size of DS?

In-class exercise (5 minutes)

1. An algorithm takes n^2 steps in total to operate on an array of size n. How many steps would the algorithm take if it were to operate on half the array of size $n/2$?
2. Has the complexity of the algorithm changed now that it is only operating only on a half?
   - If so, explain how and provide the old and new complexities (tightest bounds)
   - If not, explain why and provide the tightest complexity bound

---

Answer to 1 needs to be in terms of n.

Explanation for 2 must be specific (reference definitions of Big-O, finding tight bounds, etc.).

Searching an array

Searching an array

- Function args: `numbers` array, `target` element you are looking for
- To return: `True` if it's found, `False` if not

```
def search (numbers, target):
        …
```

- What's the simplest and most straightforward way to do this?

Search attempt #1

- Look at each element one at a time
  - Once you encounter the element, return `True`
  - If you don't encounter the element by end of loop, return `False`

```
def search(numbers, target):
    for x in numbers:
        if x == target:
            return True
    return False
```

What's the worst case?
- Element is not in the array

important steps (iterations) in worst case?
n

What if the array is huge?
- Wish we could stop early if we knew there was no hope

What if the array was sorted?

numbers	2	4	8	11	12	90

`search(numbers, 5)` should return False

No hope

```
def search(numbers, target):
    for x in numbers:
        if x > target:  # return early
            return False # if you know there's no hope
        if x == target:
            return True
    return False
```

What if the array was sorted?

```
def search(numbers, target):
    for x in numbers:
        if x > target:
            return False
        if x == target:
            return True
    return False
```

iterations may reduce for previous worst case
- Old worst case no longer applies

Worst case now?
- Target is or is higher than highest element (i.e., not in array)

iterations in worst case: still n

Linear search

- Search that takes linear time: $O(n)$
- Same complexity on sorted and unsorted arrays
- Worst case is $O(n)$
  - Average and best cases may be cheaper
  - **Usually care about worst case (but not always)**
- Can sortedness get us a better worst case cost?

```
def linear_search(numbers, target):
    for x in numbers:
        if x > target:
            return False
        if x == target:
            return True
    return False
```

Game: Guessing a number

- Your job: Think of a number between **[1, 20]**
- My job: guess your number in 5 guesses or less

- # steps to guess: 1 + 1 + 1 + 1 + 1

Game: Guessing a number (round 2)

- Your job: Think of a number between **[1, 20]**
- My job: guess your number in 5 guesses or less

1 20

- # steps to guess:

Binary search

- Type of divide-and-conquer algorithm (or decrease-and-conquer)
- Search space is reduced by half each time
- #steps for worst case:
  - I guess 10. You tell me too high (+1 step)
  - I guess 5. You tell me too low. (+1 step)
  - I guess 7. You tell me too low. (+1 step)
  - I guess 8. You say too low. (+1 step)
  - I guess 9. You say that's correct! (+1 step)
  - Total guesses = 5 $\approx \log_2 20$ (#times 20 can divide by 2 before reaching 0)

Binary search pseudocode

1. State: `start = 0` and `end = length-1`
2. Look for `midpoint` position in the array between start and end: (`end+start)/2`
3. Check if target is equal to, less than, or more than value at `midpoint`
   a. If equal: we found it, return true
   b. If less than: `end = midpoint - 1`
   c. If more than: `start = midpoint + 1`
   d. If `start` > `end`: `return false`
4. Repeat steps 2-3

0	1	2	3	4
2	5	8	9	12

Binary search code

```
def binary_search (numbers, target):
  # look for `target` between indices `low` and `high`
  def helper (low, high):
    # empty range -> not found
    if low > high: return False
    let mid = (low + high) // 2
    if numbers[mid] == target: return True
    elif numbers[mid] < target:
      return helper(mid+1, high)
    else: # numbers[mid] > target:
      return helper(low, mid-1)

  return helper(0, numbers.len()-1)
```

Binary search complexity

- Logarithmic worst-case complexity
  - $O(\log n)$
- Constant best-case complexity
  - $O(1)$ (midpoint of the whole array is the target)
  - Not what matters (best case is rare)

Another example: Looking for a word in a dictionary

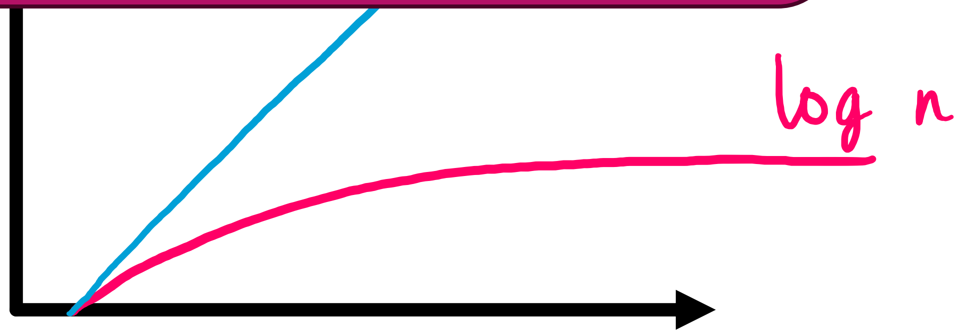
- Dictionary is sorted
- Say it 171,476 words

- Linear search would take a maximum of 171,476 steps
- Binary search takes a maximum of $\log_2 171{,}476 = 18$ steps

O(log n) is a big deal

n	10	100	1000	10K	100K	1M	10M	100M
$\log_2 n$							3.3	26.6

Considered as good as O(1) as n gets larger



Pre-condition for binary search

- Sorted array is a precondition for binary search
- We'll need a way to sort an array
  - Next time!