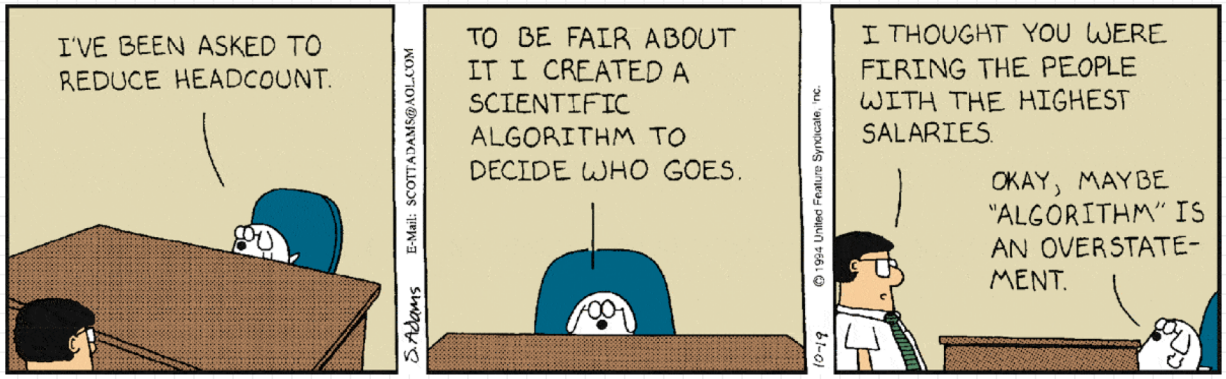


CSCI 2824: Discrete Structures

Lecture 21: Algorithms – Complexity.

Reminders

Submissions:

- Homework 7: **Fri 10/18 at noon –** Moodle (1 try)
- Quizlet : **Fri 10/18 at 8pm**

Readings:

- Ch. 3 – Algorithms
  - 3.1 Greedy Algorithms
  - 3.2 Growth of Functions

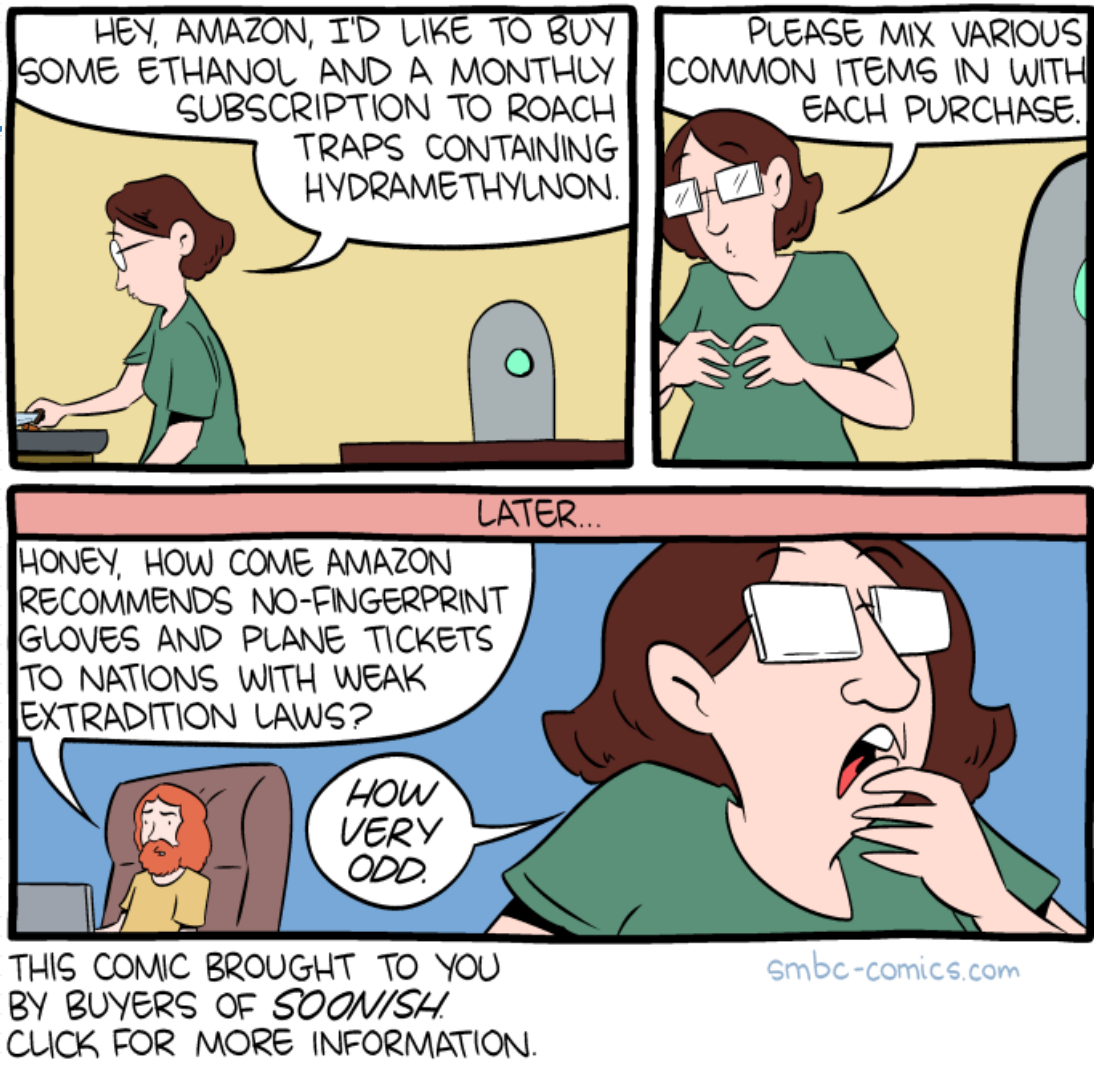
Last time

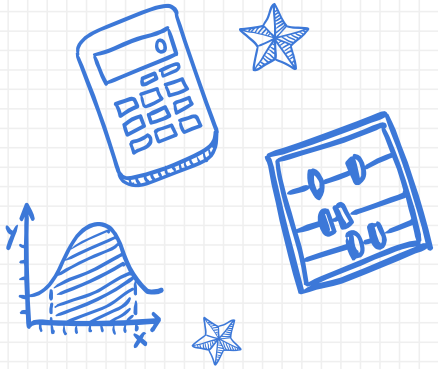
Algorithms…

- What are they?
- Special kinds: searching, sorting
- Worst case and Average case

Today:

- algorithm *complexity*
- *Greedy* Algorithms
- Growth of Functions





Algorithms

Algorithm complexity

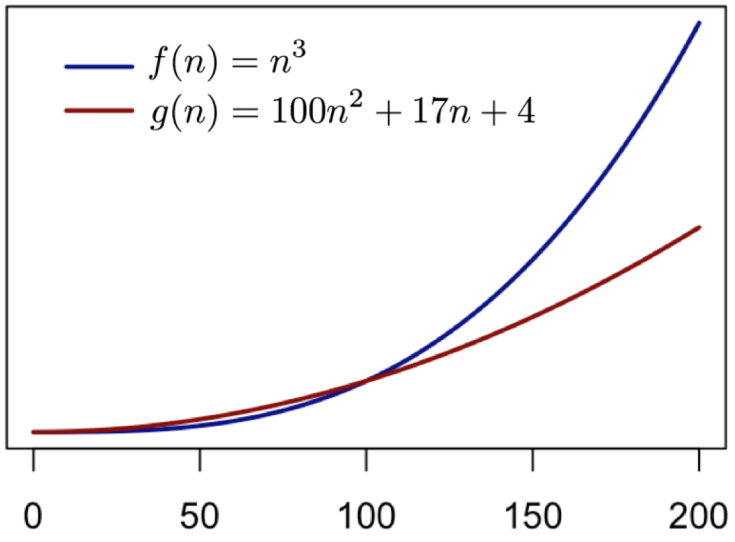
There are several conventions that allow us to talk about the efficiency of algorithms without writing out their precise operation counts.

Question: S'pose we have two algorithms that solve the same problem.

Algorithm A uses $100n^2 + 17n + 4$ operations.

Algorithm B uses n^3 operations.

Which should you use?



Algorithm complexity

There are several conventions that allow us to talk about the efficiency of algorithms without writing out their precise operation counts.

Question: S'pose we have two algorithms that solve the same problem.

Algorithm A uses $100n^2 + 17n + 4$ operations.

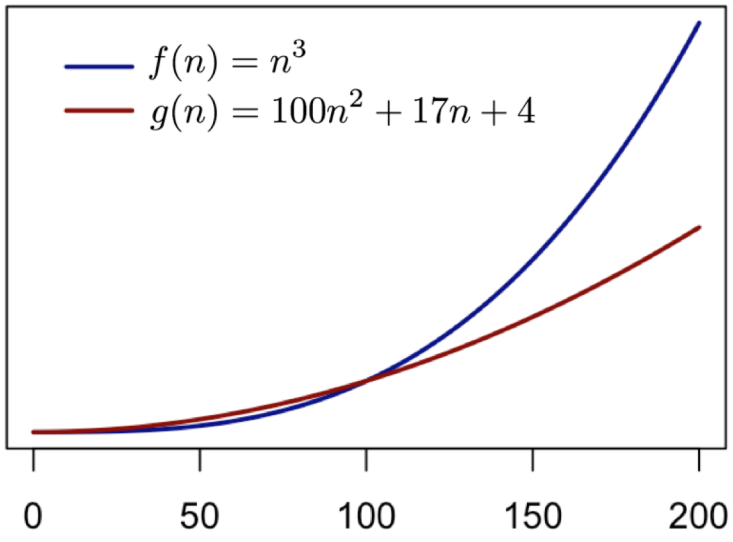
Algorithm B uses n^3 operations.

Which should you use?

Answer:

➢ For small values of n, n^3 might be less than $100n^2 + 17n + 4$ …

➢ But n^3 becomes **much** larger than $100n^2 + 17n + 4$.



Algorithm complexity

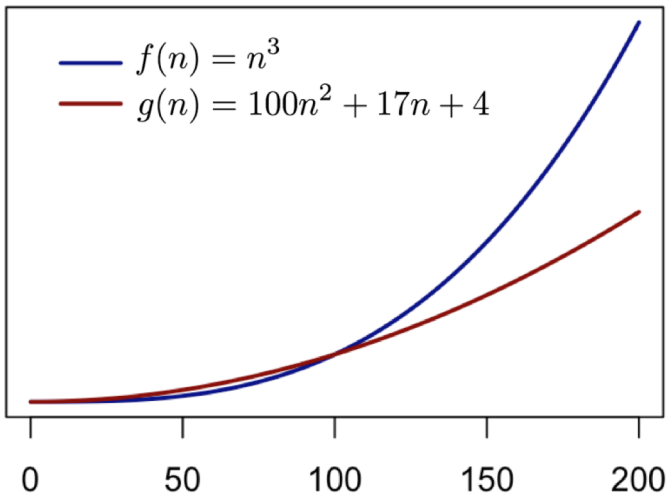
Definition: Let f and g be functions from the set of integers. We say that $f(n)$ is $O(g(n))$ if there are constants C and k such that

$$|f(n)| \leq C\,|g(n)|$$

whenever $n > k$. [[*"f(n) is big-oh of g(n)"*]]

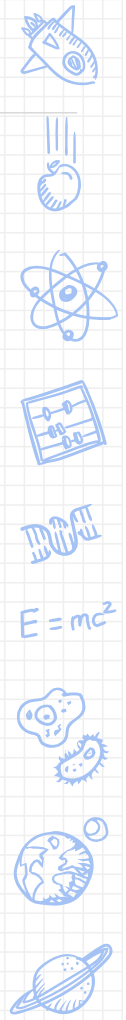
The big difference between the two is that one "is like" n^3 and the other "is like" n^2.

- We say that the complexity of Algorithm A, $100n^2 + 17n + 4$, is $O(n^2)$
- And Algorithm B's complexity, n^3, is $O(n^3)$



Algorithm complexity

Example: Show that $100n^2 + 17n + 4$ is $O(n^2)$.



Algorithm complexity

Example: Show that $100n^2 + 17n + 4$ is $O(n^2)$.

Solution:
We know that for $n > 1$, it is true that $n \leq n^2$ and $4 \leq n^2$

So for $n > 1$, we have

$$100n^2 + 17n + 4 \leq 100n^2 + 17n^2 + n^2 = 118n^2$$

So $100n^2 + 17n + 4 \leq 118n^2$ when $n > 1$,

which means that $100n^2 + 17n + 4$ is $O(n^2)$ with $C = 118$ and $k = 1$.

Algorithm complexity

Note: $f(n)$ is $O(g(n))$ means that for some C and k, $f(n)$ is bounded above by $C \cdot g(n)$

- This definition also means that $100n^2 + 17n + 4$ is $O(n^3)$ and $O(n^{1000})$...
- This is slightly unsatisfactory, because saying that $100n^2 + 17n + 4$ is $O(n^{1000})$ is in the realm of information that is true, but not very useful.

Algorithm complexity

Example: Give a big-O estimate of $h(n) = 5n^2 + n \log(n^3) - n$

Algorithm complexity

Example: Give a big-O estimate of $h(n) = 5n^2 + n \log(n^3) - n$

Solution: First note that $h(n) = 5n^2 + n \log(n^3) - n = 5n^2 + 3n \log(n) - n$

Now note that for $n > 1$, $\log n \leq n$ and $-n < 0$

So, for $n > 1$, $h(n) = 5n^2 + 3n \log n - n \leq 5n^2 + 3n^2 = 8n^2$, is $O(n^2)$

Theorem: S'pose that $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$.
Then $(f_1 + f_2)(n)$ is $O(\max(|g_1(n)|, |g_2(n)|))$

- **In words:** A sum of functions is big-O of the biggest function.

Theorem: S'pose that $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$.
Then $(f_1 f_2)(n)$ is $O(g_1(n) g_2(n))$

Algorithm complexity

- So $f(n)$ is $O(g(n))$ tells us that f grows no faster than g
- That's a nice upper bound on the growth of f
- But it would be ***really*** nice to also have a **lower bound**… to say, for example, that

$f(n)$ grows ***at least as fast as*** $h(n)$

Algorithm complexity

- So $f(n)$ is $O(g(n))$ tells us that f grows no faster than g
- That's a nice upper bound on the growth of f
- But it would be ***really*** nice to also have a **lower bound**… to say, for example, that

$f(n)$ grows ***at least as fast as*** $h(n)$

Definition: S'pose that $f(n)$ and $h(n)$ are functions from the set of integers. We say that $f(n)$ is $\Omega(h(n))$ if there are constants C and k such that

$$| f(n) | \geq C | h(n) |$$

whenever $n > k$. [[*"f(n) is big-Omega of h(n)"*]]

Algorithm complexity

Example: Give a big-Ω estimate for $100n^2 + 17n + 4$

Algorithm complexity

Example: Give a big-Ω estimate for $100n^2 + 17n + 4$

Solution:
We know that for $n > 0$, it is super true that $100n^2 + 17n + 4 \geq 100n^2$

$100n^2 + 17n + 4$ is $\Omega(n^2)$ with $C = 100$ and $k = 0$

Algorithm complexity

Example: Give a big-Ω estimate for $100n^2 + 17n + 4$

Solution:

We know that for $n > 0$, it is super true that $100n^2 + 17n + 4 \geq 100n^2$

$100n^2 + 17n + 4$ is $O(n^2)$ with $C = 100$ and $k = 0$

$100n^2 + 17n + 4$ is $\Omega(n^2)$ with $C = 100$ and $k = 0$

Fun fact: We see that $100n^2 + 17n + 4$ is both $O(n^2)$ and $\Omega(n^2)$

⇒ it grows no faster than a constant times n^2, and no slower than some other constant times n^2

⇒ **Definition:** when this happens, we say that $100n^2 + 17n + 4$ is $\Theta(n^2)$

[[*"big-Theta of n^2"*]]

Algorithm complexity

Big-Θ estimates of a function are the holy grail!

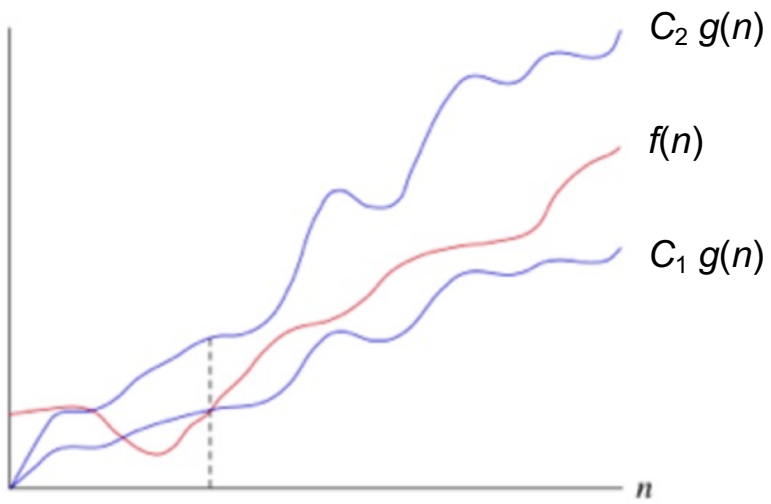
- They say we know everything about the growth of a function
- It's smaller than *this*, but it's larger than *that*.

Definition: If $f(n)$ and $g(n)$ are both $\Theta(h(n))$, then we say that f and g have **order** $h(n)$, and that f and g are the same order.

Growth of Functions

Definition: If *f(n)* is both *O* (*g*(*n*)) and Ω(*g*(*n*)) then we say *f(n)* is Θ(*g(n)*), and also say that *f(n)* is of **order** *g(n)*.

[[*"f(n) is big-Theta of g(n)"*]]



Algorithm complexity

Example: Show that $h(n) = 2n^2 + 5n \log n$ is $\Theta(n^2)$

Solution: Two steps: first, show $h(n)$ is $O(n^2)$; then, show $h(n)$ is $\Omega(n^2)$

Showing $h(n)$ is $O(n^2)$ …

Algorithm complexity

Example: Show that $h(n) = 2n^2 + 5n \log n$ is $\Theta(n^2)$

Solution: Two steps: first, show $h(n)$ is $O(n^2)$; then, show $h(n)$ is $\Omega(n^2)$

Showing $h(n)$ is $O(n^2)$ …

Note that for $n > 1$, $\log n \leq n$

$\Rightarrow n \log n \leq n^2$

$\Rightarrow h(n) = 2n^2 + 5n \log n \leq 2n^2 + 5n^2 = 7n^2$

$\Rightarrow$ So we have $h(n)$ is $O(n^2)$ (with $C = 7$ and $k = 1$)

Algorithm complexity

Example: Show that $h(n) = 2n^2 + 5n \log n$ is $\Theta(n^2)$

Solution: Two steps: first, show $h(n)$ is $O(n^2)$; then, show $h(n)$ is $\Omega(n^2)$

Showing $h(n)$ is $\Omega(n^2)$ …

Algorithm complexity

Example: Show that $h(n) = 2n^2 + 5n \log n$ is $\Theta(n^2)$

Solution: Two steps: first, show $h(n)$ is $O(n^2)$; then, show $h(n)$ is $\Omega(n^2)$

Showing $h(n)$ is $\Omega(n^2)$ …

Easier: $h(n) = 2n^2 + 5n \log n \geq 2n^2$ for $n > 1$

$\Rightarrow$ So we have $h(n)$ is $\Omega(n^2)$ (with $C = 2$ and $k = 1$)

Since h(n) is both $O(n^2)$ and $\Omega(n^2)$, it is $\Theta(n^2)$ □

Greedy algorithms

Many algorithms are designed to solve ***optimization problems***.

Goal: To find a solution that maximizes or minimizes some ***objective function***.

Applications: Find a route between two cities that minimizes distance (or time travelled, or elevation gain, …).
Encode a message using the fewest bits possible.

Greedy algorithms select the best choice at each step.

Note: they are *not* guaranteed to find an optimal solution. You must check once a solution is found.

Greedy algorithms

Task: Consider making *n* cents change, using quarters, dimes, nickels and pennies, using the fewest total number of coins.
Greedy strategy: At each step, choose the largest denomination coin possible to add to the pile, so as not to exceed *n* cents.

Example: S'pose we want to make change for 67 cents.

1. Select a quarter (25 cents)
2. Select another quarter (50 cents)
3. Select a dime (60 cents)
4. Select a nickel (65 cents)
5. Select a penny (66 cents)
6. Select another penny (67 cents)

{c[0], c[1], c[2], …, c[N]} = coin denominations, eg...

```
def greedy (amt, c):
    total = amt
    d = []                  # initialize, to count coins of each denomination
    n_thiscoin = 0          # initialize, to count the current coin
    for thiscoin in range(0, len(c)):
        while (total ≥ c[thiscoin]):
            n_thiscoin = n_thiscoin + 1
            total = total - c[thiscoin]    # add another one of this coin
        d.append(n_thiscoin)
        n_thiscoin = 0                     # reset counter to 0
    return (d)
```

Greedy Algorithms

Fun Fact: If we have **quarters**, **dimes**, **nickels** and **pennies** available, then this algorithm is optimal.

Fun? Fact: If we only have **quarters**, **dimes** and **pennies** available (no nickels), then this algorithm is ***not*** optimal.

Example: If we wanted to make change for 30 cents, using no nickels, then we would:

1. take 1 quarter
2. take 5 pennies

… But it would be ***optimal*** to take 3 dimes instead.

Matrices and matrix operations

- **Linear algebra** is the workhorse of computational science.
- In scientific/engineering computing, a huge amount of computing time is spent on matrix operations.
- Applications of matrices are broad, but they were invented for **a simple purpose**: *to make solving systems of equations cleaner.*

$$\begin{aligned} 3x + 4y + 5z &= 1 \\ 2x + 8y + 3z &= 2 \\ 4x + 2y + 2z &= 3 \end{aligned} \quad \Leftrightarrow \quad \underbrace{\begin{bmatrix} 3 & 4 & 5 \\ 2 & 8 & 3 \\ 4 & 2 & 2 \end{bmatrix}}_{\text{Matrix}} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \underbrace{\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}}_{\text{Vectors}}$$

Matrices and matrix operations

$$\begin{aligned} 3x + 4y + 5z &= 1 \\ 2x + 8y + 3z &= 2 \\ 4x + 2y + 2z &= 3 \end{aligned} \quad \Leftrightarrow \quad \begin{bmatrix} 3 & 4 & 5 \\ 2 & 8 & 3 \\ 4 & 2 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

- Rectangular thing is a **matrix**, tall skinny things are **vectors**
- **Definition:** A matrix with *m* rows and *n* columns has **dimensions** $m \times n$
- **Definition:** A vector with *n* entries has **length** *n*
- **Notation:** Matrices are represented by capital letters, like *A* and *M*.
  Vectors are represented by lowercase letters like **x** and **b**
  (often bold-faced)
- **Example:** The above **matrix equation** could be written as $A\mathbf{x} = \mathbf{b}$

Matrices and matrix operations

- Matrices and vectors can be **added** and **multiplied** (but not divided)
- **Definition:** The **sum** of matrices *A* and *B* is the matrix obtained by adding the corresponding entries of each matrix together
- **Example:**

$$\begin{bmatrix} 3 & 4 & 5 \\ 2 & 8 & 3 \\ 4 & 2 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 4 & 6 & 8 \\ 6 & 13 & 9 \\ 11 & 10 & 11 \end{bmatrix}$$

- **Note:** Only makes sense if *A* and *B* have the **same** dimensions!
- **Notation:** We refer to the entry in the i^{th} row and j^{th} column of the matrix *A* as a_{ij}, or $A[i, j]$.

Matrices and matrix operations

Complexity of matrix addition:

- Straightforward: We add each pair of entries.

  ⇒ for two $m \times n$ matrices, there are $m*n$ entries, so $m*n$ additions

  ⇒ for square matrices of size $n \times n$, that's n^2 additions, so this is $O(n^2)$

Matrices and matrix operations

Complexity of matrix addition:

- Using the slick pseudocode method (for adding two square matrices):

```
def matrixAdd (A, B):
      S = "0"                          # initialize as n × n zero matrix
      for i in 1,num_rows:
            for j in 1,num_cols:
                  S[i,j] = A[i,j] + B[i,j]
      return (S)
```

Turn loops into sums and count up the basic operations:

Complexity:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2$$

Matrices and matrix operations

- Matrices can also **multiply** vectors, resulting in a new vector.

$$\begin{aligned} 3x + 4y + 5z &= 1 \\ 2x + 8y + 3z &= 2 \\ 4x + 2y + 2z &= 3 \end{aligned} \quad \Leftrightarrow \quad \begin{bmatrix} 3 & 4 & 5 \\ 2 & 8 & 3 \\ 4 & 2 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

- This operation is defined directly out of the analogy between matrix equations and linear systems of equations.

$$\begin{aligned} 3x + 4y + 5z \\ 2x + 8y + 3z \\ 4x + 2y + 2z \end{aligned} \quad \Leftrightarrow \quad \begin{bmatrix} 3 & 4 & 5 \\ 2 & 8 & 3 \\ 4 & 2 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Matrices and matrix operations

- Think of it as taking the vector and setting it down on top of the matrix.

$$\begin{bmatrix} 3 & 4 & 5 \\ 2 & 8 & 3 \\ 4 & 2 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 3\cdot 1 + 4\cdot 2 + 5\cdot 3 \\ 2\cdot 1 + 8\cdot 2 + 3\cdot 3 \\ 4\cdot 1 + 2\cdot 2 + 2\cdot 3 \end{bmatrix} = \begin{bmatrix} 26 \\ 27 \\ 14 \end{bmatrix}$$

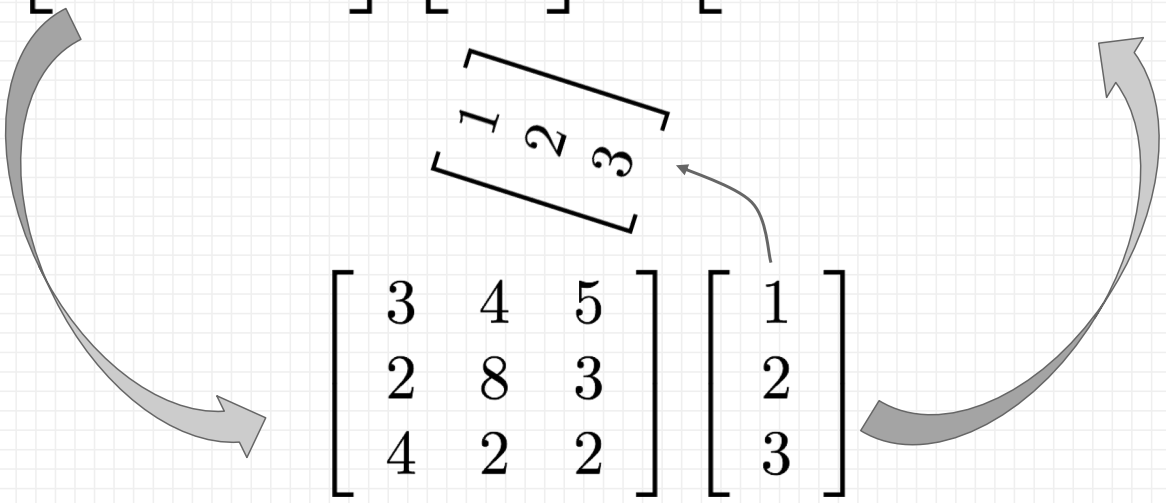
$$\begin{bmatrix} 3 & 4 & 5 \\ 2 & 8 & 3 \\ 4 & 2 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Important rule: This means that the length of the vector **must equal** the number of columns of the matrix.

Matrices and matrix operations

- Think of it as taking the vector and setting it down on top of the matrix.

$$\begin{bmatrix} 3 & 4 & 5 \\ 2 & 8 & 3 \\ 4 & 2 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 3\cdot 1 + 4\cdot 2 + 5\cdot 3 \\ 2\cdot 1 + 8\cdot 2 + 3\cdot 3 \\ 4\cdot 1 + 2\cdot 2 + 2\cdot 3 \end{bmatrix} = \begin{bmatrix} 26 \\ 27 \\ 14 \end{bmatrix}$$

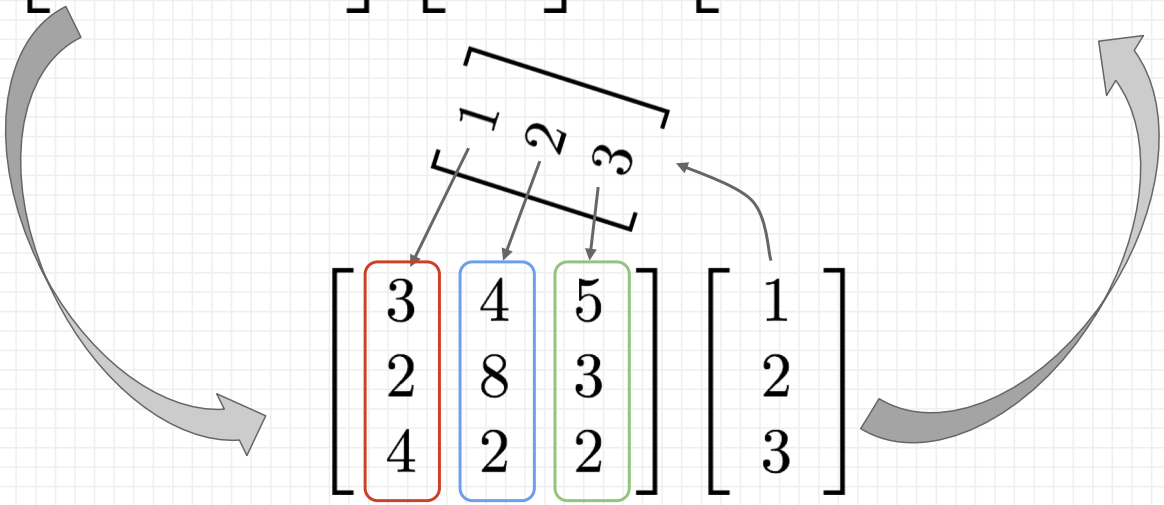


Important rule: This means that the length of the vector **must equal** the number of columns of the matrix.

Matrices and matrix operations

- Think of it as taking the vector and setting it down on top of the matrix.

$$\begin{bmatrix} 3 & 4 & 5 \\ 2 & 8 & 3 \\ 4 & 2 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 3 \cdot 1 + 4 \cdot 2 + 5 \cdot 3 \\ 2 \cdot 1 + 8 \cdot 2 + 3 \cdot 3 \\ 4 \cdot 1 + 2 \cdot 2 + 2 \cdot 3 \end{bmatrix} = \begin{bmatrix} 26 \\ 27 \\ 14 \end{bmatrix}$$



Important rule: This means that the length of the vector **must equal** the number of columns of the matrix.

Matrices and matrix operations

Example: Compute *A***x**, where

$$A = \begin{bmatrix} 1 & 3 \\ -2 & 4 \\ 1 & 5 \end{bmatrix}, \text{and } \mathbf{x} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 \\ -2 & 4 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 2 + 3 \cdot 1 \\ -2 \cdot 2 + 4 \cdot 1 \\ 1 \cdot 2 + 5 \cdot 1 \end{bmatrix} = \begin{bmatrix} 2 + 3 \\ -4 + 4 \\ 2 + 5 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \\ 7 \end{bmatrix}$$

Matrices and matrix operations

Pseudocode and complexity: intuition and estimation – counting multiplications/additions, what is a rough estimate of the complexity?

$$\begin{bmatrix} 1 & 3 \\ -2 & 4 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 2 + 3 \cdot 1 \\ -2 \cdot 2 + 4 \cdot 1 \\ 1 \cdot 2 + 5 \cdot 1 \end{bmatrix} = \begin{bmatrix} 2 + 3 \\ -4 + 4 \\ 2 + 5 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \\ 7 \end{bmatrix}$$

Matrices and matrix operations

Pseudocode and complexity: intuition and estimation – counting multiplications/additions, what is a rough estimate of the complexity?

```
In [244]: y = []                    # initialize output vector
     ...: n = len(A)
     ...: for i in range(0,n):
     ...:   y.append(A[i][0]*x[0])
     ...:   for j in range(1,n):
     ...:       y[i] = y[i] + A[i][j]*x[j]
```

$$\begin{bmatrix} 1 & 3 \\ -2 & 4 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 2 + 3 \cdot 1 \\ -2 \cdot 2 + 4 \cdot 1 \\ 1 \cdot 2 + 5 \cdot 1 \end{bmatrix} = \begin{bmatrix} 2 + 3 \\ -4 + 4 \\ 2 + 5 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \\ 7 \end{bmatrix}$$

Matrices and matrix operations

Pseudocode and complexity: intuition and estimation – counting multiplications/additions, what is a rough estimate of the complexity?

```
In [244]: y = []                    # initialize output vector
     ...: n = len(A)
     ...: for i in range(0,n):
     ...:   y.append(A[i][0]*x[0])
     ...:   for j in range(1,n):
     ...:       y[i] = y[i] + A[i][j]*x[j]
```

- Count additions and multiplications. (Usually, we count FLOPs (floating point operations) instead.)
- **Complexity:** $= \sum_{i=1}^{n} 1 \sum_{j=1}^{n} 2 = \sum_{i=1}^{n} 2n = 2n^2$

Matrix Multiplication

Multiplying matrices: think of it as doing a few matrix-vector multiplications

$$\begin{bmatrix} 1 & 3 & -1 \\ -2 & 4 & 0 \\ 1 & 5 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \\ 1 & 0 & -2 \end{bmatrix} = ?$$

$$AB = A[\mathbf{b}_1 \ \mathbf{b}_2 \ \mathbf{b}_3] = [A\mathbf{b}_1 \ A\mathbf{b}_2 \ A\mathbf{b}_3]$$

$$\begin{bmatrix} 1 & 3 & -1 \\ -2 & 4 & 0 \\ 1 & 5 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \\ 1 & 0 & -2 \end{bmatrix} = \begin{bmatrix} 1\cdot 2 + 3\cdot 3 + -1\cdot 1 & - & - \\ - & - & - \\ - & - & - \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 & -1 \\ -2 & 4 & 0 \\ 1 & 5 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \\ 1 & 0 & -2 \end{bmatrix} = \begin{bmatrix} 1\cdot 2+3\cdot 3+-1\cdot 1 & \_ & \_ \\ \_ & \_ & \_ \\ \_ & \_ & \_ \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 & -1 \\ -2 & 4 & 0 \\ 1 & 5 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \\ 1 & 0 & -2 \end{bmatrix} = \begin{bmatrix} 10 & \_ & \_ \\ -2\cdot 2+4\cdot 3+0\cdot 1 & \_ & \_ \\ \_ & \_ & \_ \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 & -1 \\ -2 & 4 & 0 \\ 1 & 5 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \\ 1 & 0 & -2 \end{bmatrix} = \begin{bmatrix} 1\cdot 2+3\cdot 3+-1\cdot 1 & \_ & \_ \\ \_ & \_ & \_ \\ \_ & \_ & \_ \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 & -1 \\ -2 & 4 & 0 \\ 1 & 5 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \\ 1 & 0 & -2 \end{bmatrix} = \begin{bmatrix} 10 & \_ & \_ \\ -2\cdot 2+4\cdot 3+0\cdot 1 & \_ & \_ \\ \_ & \_ & \_ \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 & -1 \\ -2 & 4 & 0 \\ 1 & 5 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \\ 1 & 0 & -2 \end{bmatrix} = \begin{bmatrix} 10 & \_ & \_ \\ 8 & \_ & \_ \\ 1\cdot 2+5\cdot 3+2\cdot 1 & \_ & \_ \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 & -1 \\ -2 & 4 & 0 \\ 1 & 5 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \\ 1 & 0 & -2 \end{bmatrix} = \begin{bmatrix} 10 & -2 & 8 \\ 8 & -6 & 8 \\ 19 & -4 & 6 \end{bmatrix}$$

- **Question:** What must be the dimensions of *A* and *B* for this to work?

$$\begin{bmatrix} 1 & 3 & -1 \\ -2 & 4 & 0 \\ 1 & 5 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \\ 1 & 0 & -2 \end{bmatrix} = \begin{bmatrix} 10 & -2 & 8 \\ 8 & -6 & 8 \\ 19 & -4 & 6 \end{bmatrix}$$

- **Question:** What must be the dimensions of *A* and *B* for this to work?

  Answer: Need the # columns of *A* to match the # rows of *B*.

- **Question:** What are the dimensions of $C = AB$?

$$\begin{bmatrix} 1 & 3 & -1 \\ -2 & 4 & 0 \\ 1 & 5 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 3 & -1 & 2 \\ 1 & 0 & -2 \end{bmatrix} = \begin{bmatrix} 10 & -2 & 8 \\ 8 & -6 & 8 \\ 19 & -4 & 6 \end{bmatrix}$$

- **Question:** What must be the dimensions of *A* and *B* for this to work?

  Answer: Need the # columns of *A* to match the # rows of *B*.

- **Question:** What are the dimensions of *C = AB* ?

  Answer: *C = AB* will have the same # rows as *A* and # columns as *B*.

- **Summary:** If *A* is *n x k* and *B* is *k x m* then *C = AB* is *n x m*.

Complexity: Multiply *A* and *B*, both *n x n* matrices.

$$AB = A[\mathbf{b}_1 \ \mathbf{b}_2 \ \mathbf{b}_3] = [A\mathbf{b}_1 \ A\mathbf{b}_2 \ A\mathbf{b}_3]$$

Complexity: Multiply *A* and *B*, both *n x n* matrices.

$$AB = A[\mathbf{b}_1 \ \mathbf{b}_2 \ \mathbf{b}_3] = [A\mathbf{b}_1 \ A\mathbf{b}_2 \ A\mathbf{b}_3]$$

- Each column of $C = AB$ is a "mat-vec"
- We saw these each require ~$2n^2$ FLOPs
- And we have *n* columns to do

⇒ Total mat-mat is ~n x $2n^3$ = $2n^3$ FLOPs

Summary:

- Matrix addition is $O(n^2)$
- Matrix-vector multiplication (mat-vec) is $O(n^2)$
- Matrix-matrix multiplication (mat-mat) is $O(n^3)$

Algorithm complexity and matrix operations

Extra special FYOG: There is a special kind of a square matrix called **lower-triangular**, where all of the elements above the **main diagonal** are 0.

- **Example:**

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 4 & 0 \\ 1 & 5 & 2 \end{bmatrix}$$

S'pose we want to compute *L***x**, where **x** is an appropriately-sized vector. Any time an entry of **x** hits one of the upper 0s, we already know the result will be 0. So we don't want to waste time computing those multiplications.

- **TODO:** Modify your mat-vec code to skip the unnecessary multiplications, assuming a lower-triangular matrix is used.
- **TODO:** Determine the complexity of the new algorithm when *L* is *n* x *n*.

Algorithm Complexity

Recap:

- We learned about estimating the **complexity of algorithms**
- We learned about estimating the **growth rates of functions**

… because that's useful to see **how (in)efficient an algorithm is**, depending on the size of the input

Next time:

- More Matrices!

Algorithm complexity

FYOG: Show that $1^k + 2^k + \ldots + n^k$ is $O(n^{k+1})$

FYOG: Show that $\log(n^2 + 1)$ and $\log n$ are the same order

FYOG: Show that $\log_{10} n$ and $\log_2 n$ are the same order

FYOG: Give a big-Ω estimate for

$$h(n) = 5n^2 + n \log(n^3) - n$$