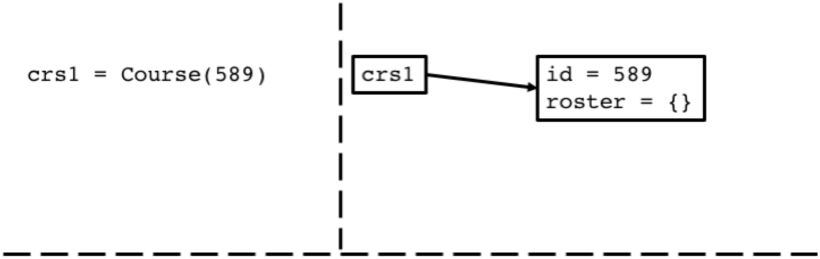
COMPSCI 590N
Lecture 7: Complexity and Numerical Linear Algebra 1

Roy J. Adams

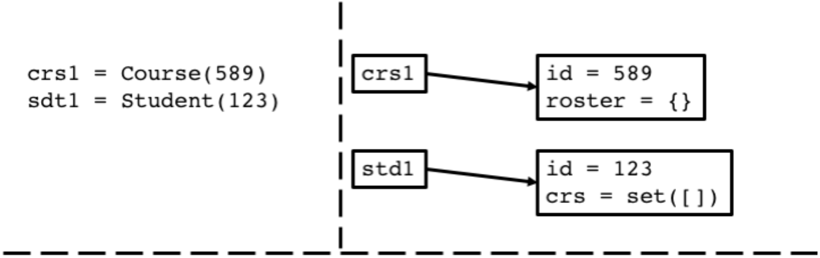
College of Information and Computer Sciences
University of Massachusetts Amherst

Outline

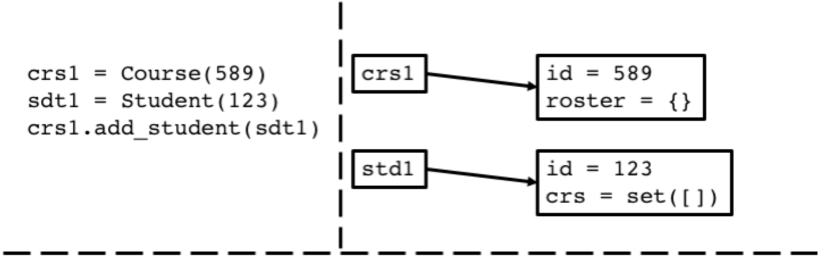
Problem 1



Problem 1

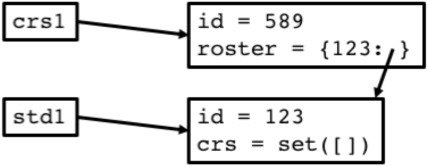

crs1 = Course(589)
sdt1 = Student(123)
crs1
id = 589
roster = {}
std1
id = 123
crs = set([])

Problem 1


crs1 = Course(589)
sdt1 = Student(123)
crs1.add_student(sdt1)
crs1
id = 589
roster = {}
std1
id = 123
crs = set([])

Problem 1

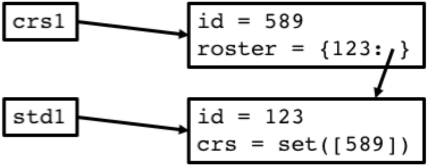
```
crs1 = Course(589)
sdt1 = Student(123)
crs1.add_student(sdt1)
```




```
self.roster[student.id] = student
```

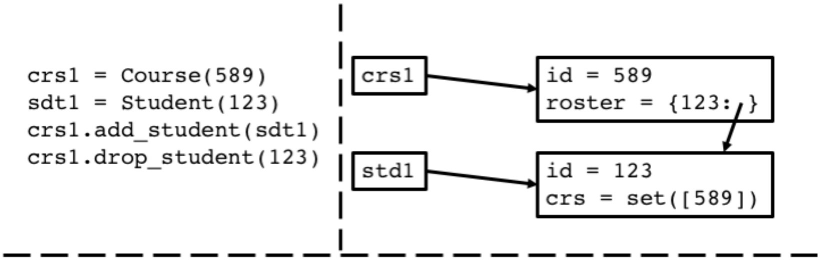
Problem 1

```
crs1 = Course(589)
sdt1 = Student(123)
crs1.add_student(sdt1)
```



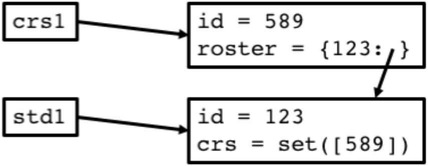

```
self.roster[student.id] = student
student.roster.add(self.id)
```

Problem 1


crs1 = Course(589)
sdt1 = Student(123)
crs1.add_student(sdt1)
crs1.drop_student(123)
crs1
id = 589
roster = {123: }
std1
id = 123
crs = set([589])

Problem 1

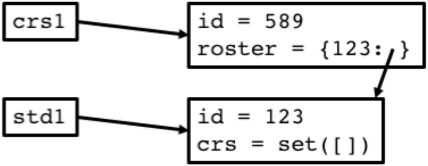
```
crs1 = Course(589)
sdt1 = Student(123)
crs1.add_student(sdt1)
crs1.drop_student(123)
```




```
student = self.roster[student_id]
```

Problem 1

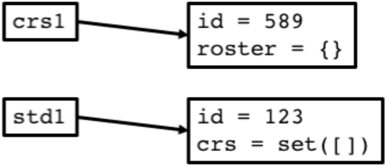
```
crs1 = Course(589)
sdt1 = Student(123)
crs1.add_student(sdt1)
crs1.drop_student(123)
```




```
student = self.roster[student_id]
student.crs.remove(self.id)
```

Problem 1

```
crs1 = Course(589)
sdt1 = Student(123)
crs1.add_student(sdt1)
crs1.drop_student(123)
```




```
student = self.roster[student_id]
student.crs.remove(self.id)
del self.roster[student_id]
```

Outline

Views

A `view` of an array is another `ndarray` object that shares the same underlying memory locations.

Views

A `view` of an array is another `ndarray` object that shares the same underlying memory locations.

```
>>> A = np.ones((4,5))
>>> B = A.view()
>>> B is A
False
>>> B.base is A
True
```

Views

A `view` of an array is another `ndarray` object that shares the same underlying memory locations.

```
>>> A = np.ones((4,5))
>>> B = A.view()
>>> B is A
False
>>> B.base is A
True
```

- `A` and `B` are not the same object, hence `A is B` returns `False`.

Views

A `view` of an array is another `ndarray` object that shares the same underlying memory locations.

```
>>> A = np.ones((4,5))
>>> B = A.view()
>>> B is A
False
>>> B.base is A
True
```

- `A` and `B` are not the same object, hence `A is B` returns `False`.
- The attribute `base` points to the array from which a view was created, hence `B.base is A` returns `True`.

Views

Many functions will try to create views if they can, **but will silently create copies if they can't.** Examples of such operations/functions are:

Views

Many functions will try to create views if they can, **but will silently create copies if they can't.** Examples of such operations/functions are:

- Slicing

Views

Many functions will try to create views if they can, **but will silently create copies if they can't.** Examples of such operations/functions are:

- Slicing
- transpose

Views

Many functions will try to create views if they can, **but will silently create copies if they can't.** Examples of such operations/functions are:

- Slicing
- transpose
- reshape

Views

```
>>> A = np.ones((4,5))
>>> B = A.transpose()
>>> B.base is A
True
>>> C = B.reshape((5,4)) # Creates another view
>>> C.base is A
True
>>> D = B.reshape((2,10)) # Creates a copy!
>>> D.base is A
False
```

Views

- There are rules for when a view will be made, however, they are not well documented and rely on an understanding of the underlying storage.

Views

- There are rules for when a view will be made, however, they are not well documented and rely on an understanding of the underlying storage.
- The moral of the story is: be careful when modifying the contents of an array directly.

Views

- There are rules for when a view will be made, however, they are not well documented and rely on an understanding of the underlying storage.
- The moral of the story is: be careful when modifying the contents of an array directly.
- When in doubt, create a copy!

Outline

Computational Complexity

- In our discussion of numerical algorithms so far, we have obliquely discussed the speed of our algorithms.

Computational Complexity

- In our discussion of numerical algorithms so far, we have obliquely discussed the speed of our algorithms.
- We need a formal language for discussing the speed of an algorithm.

Computational Complexity

- In our discussion of numerical algorithms so far, we have obliquely discussed the speed of our algorithms.
- We need a formal language for discussing the speed of an algorithm.
- The mathematical language used by computer scientists and mathematicians is called *complexity*.

Computational Complexity

High level idea: The complexity of an algorithm is the number of *basic operations* used by the algorithm as a function of the input size.

Computational Complexity

High level idea: The complexity of an algorithm is the number of *basic operations* used by the algorithm as a function of the input size.

Basic operations

A basic operation is any computation that has a constant runtime as a function of the input size. Think of basic operations as the building blocks of algorithms.

Computational Complexity

High level idea: The complexity of an algorithm is the number of *basic operations* used by the algorithm as a function of the input size.

Basic operations

A basic operation is any computation that has a constant runtime as a function of the input size. Think of basic operations as the building blocks of algorithms.

Examples include: arithmetic operations, basic functions, logical comparisons, basic indexing, and assignment.

Array Scanning

Let's consider an example: Finding the maximum value in a list of numbers.

Array Scanning

Let's consider an example: Finding the maximum value in a list of numbers.

```
def max(A):
    m = -inf
    for a in A:
        if a > m:
            m = a
    return m
```

Array Scanning

Let's consider an example: Finding the maximum value in a list of numbers.

```
def max(A):
    m = -inf
    for a in A:
        if a > m:
            m = a
    return m
```

- What are the basic operations used in this algorithm?

Array Scanning

Let's consider an example: Finding the maximum value in a list of numbers.

```
def max(A):
    m = -inf
    for a in A:
        if a > m:
            m = a
    return m
```

- What are the basic operations used in this algorithm?
- If `A` has length n, how many basic operations will be performed by `max`?

Array Scanning

Let's consider an example: Finding the maximum value in a list of numbers.

```
def max(A):
    m = -inf
    for a in A:
        if a > m:
            m = a
    return m
```

- What are the basic operations used in this algorithm?
- If `A` has length n, how many basic operations will be performed by `max`?
- Aside from n, does the run time of `max` depend on the contents of `A`?

Worst case analysis

In the previous, example, `max` didn't depend on the contents of `A` beyond its size. This is not always the case.

Worst case analysis

In the previous, example, `max` didn't depend on the contents of `A` beyond its size. This is not always the case.

```
def find_first_zero(A):
    i = 0
    for a in A:
        if a == 0:
            return i
        else:
            i += 1
    return None
```

Worst case analysis

In the previous, example, `max` didn't depend on the contents of `A` beyond its size. This is not always the case.

```
def find_first_zero(A):
    i = 0
    for a in A:
        if a == 0:
            return i
        else:
            i += 1
    return None
```

- What are the basic operations used in this algorithm?

Worst case analysis

In the previous, example, `max` didn't depend on the contents of `A` beyond its size. This is not always the case.

```
def find_first_zero(A):
    i = 0
    for a in A:
        if a == 0:
            return i
        else:
            i += 1
    return None
```

- What are the basic operations used in this algorithm?
- If `A` has length n, how many basic operations will be performed by `find_first_zero`?

Worst case analysis

In the previous, example, `max` didn't depend on the contents of `A` beyond its size. This is not always the case.

```
def find_first_zero(A):
    i = 0
    for a in A:
        if a == 0:
            return i
        else:
            i += 1
    return None
```

- What are the basic operations used in this algorithm?
- If `A` has length n, how many basic operations will be performed by `find_first_zero`?
- It is standard practice to assume the worst possible contents of `A`. This is called **worst-case analysis**.

Binary Search

Problem: Given a **sorted** list of unique numbers, `A`, and a number of interest, `x`, find the position of `x` in `A`. Return `None` if `x` is not in `A`.

Binary Search

Problem: Given a **sorted** list of unique numbers, `A`, and a number of interest, `x`, find the position of `x` in `A`. Return `None` if `x` is not in `A`.

```
def linear_search(A,x):
    i = 0
    for a in A:
        if a == x:
            return i
        else:
            i += 1
    return None
```

Binary Search

Problem: Given a **sorted** list of unique numbers, `A`, and a number of interest, `x`, find the position of `x` in `A`. Return `None` if `x` is not in `A`.

```
def linear_search(A,x):
    i = 0
    for a in A:
        if a == x:
            return i
        else:
            i += 1
    return None
```

- What is the worst case?

Binary Search

Problem: Given a **sorted** list of unique numbers, `A`, and a number of interest, `x`, find the position of `x` in `A`. Return `None` if `x` is not in `A`.

```
def linear_search(A,x):
    i = 0
    for a in A:
        if a == x:
            return i
        else:
            i += 1
    return None
```

- What is the worst case?
- If `A` has length n, what is the worst case number of operations performed by `linear_search`?

Binary Search

Problem: Given a **sorted** list of unique numbers, `A`, and a number of interest, `x`, find the position of `x` in `A`. Return `None` if `x` is not in `A`.

```
def linear_search(A,x):
    i = 0
    for a in A:
        if a == x:
            return i
        else:
            i += 1
    return None
```

- What is the worst case?
- If `A` has length n, what is the worst case number of operations performed by `linear_search`?
- What information is not being used by `linear_search`?

Binary Search

Instead, we can use a more sophisticated algorithm called **binary search**.

Binary Search

Instead, we can use a more sophisticated algorithm called **binary search**.

Binary Search: Basic Idea

- Look at the item in the middle of the list, call this item m.

Binary Search

Instead, we can use a more sophisticated algorithm called **binary search**.

Binary Search: Basic Idea

- Look at the item in the middle of the list, call this item m.
- We know that everything before m is less than m and everything after is greater than m.

Binary Search

Instead, we can use a more sophisticated algorithm called **binary search**.

Binary Search: Basic Idea

- Look at the item in the middle of the list, call this item m.
- We know that everything before m is less than m and everything after is greater than m.
- Therefore, if $x > m$, we can eliminate the first half of the list and if $x < m$ we can eliminate the second half.

Binary Search

Instead, we can use a more sophisticated algorithm called **binary search**.

Binary Search: Basic Idea

- Look at the item in the middle of the list, call this item m.
- We know that everything before m is less than m and everything after is greater than m.
- Therefore, if $x > m$, we can eliminate the first half of the list and if $x < m$ we can eliminate the second half.
- Apply this idea recursively.

Binary Search

Animation

Binary Search

The following code implements binary search in Python.

Binary Search

The following code implements binary search in Python.

```
def binary_search(A,x):
    cur_min_idx = 0
    cur_max_idx = len(A) - 1
    while cur_min_idx <= cur_max_idx:
        cur_split = cur_min_idx + np.floor((cur_max_idx
            - cur_min_idx)/2)
        if A[cur_split] > x:
            cur_min_idx = cur_split + 1
        elif A[cur_split] < x:
            cur_max_idx = cur_split - 1
        else:
            return cur_split
    return None
```

Binary Search

- What is the worst case?

Binary Search

- What is the worst case?
- If `A` has length n, what is the worst case number of operations performed by `binary_search`?

Binary Search

- What is the worst case?
- If `A` has length n, what is the worst case number of operations performed by `binary_search`?
  - In each iteration, we are dividing the list in half so `binary_search` will take $\log(n)$ iterations.

Big O Notation

Computer scientists and mathematicians have a formal notation for discussing the run time of an algorithm as a function of the algorithm's input size.

Big O Notation

Computer scientists and mathematicians have a formal notation for discussing the run time of an algorithm as a function of the algorithm's input size.

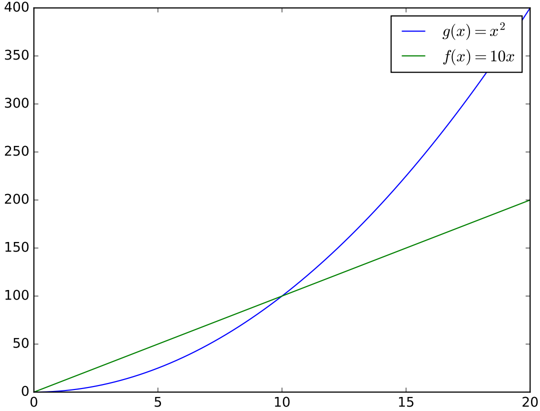
Big O Notation

Let f and g be functions, then $f(x) = \mathcal{O}(g(x))$ if and only if there exists a constant M and x_0 such that

$$|f(x)| \leq M|g(x)| \text{ for all } x \geq x_o$$

Intuitively, this means that for large numbers $g(x)$ is always greater than $f(x)$, up to a constant.

Big O Notation



Takeaways

- An algorithm's **complexity** is a measure of its run-time as a function of the input size, ignoring constant factors.

Takeaways

- An algorithm's **complexity** is a measure of its run-time as a function of the input size, ignoring constant factors.
- It is typical to consider the **worst-case** input of a fixed size.

Takeaways

- An algorithm's **complexity** is a measure of its run-time as a function of the input size, ignoring constant factors.
- It is typical to consider the **worst-case** input of a fixed size.
- Big-O notation is used to formalize this concept so that algorithms can be rigorously compared.

Outline

Intro

- Numerical Linear Algebra is the study of efficient algorithms for performing linear algebra computations.

Intro

- Numerical Linear Algebra is the study of efficient algorithms for performing linear algebra computations.
- Linear algebra computations are at the core **many** algorithms. For example:

Intro

- Numerical Linear Algebra is the study of efficient algorithms for performing linear algebra computations.
- Linear algebra computations are at the core **many** algorithms. For example:
  - Linear regression uses matrix multiplication and inversion.

Intro

- Numerical Linear Algebra is the study of efficient algorithms for performing linear algebra computations.
- Linear algebra computations are at the core **many** algorithms. For example:
  - Linear regression uses matrix multiplication and inversion.
  - PageRank, the algorithm that spawned Google, was fundamentally calculating eigenvalues.

Intro

- Numerical Linear Algebra is the study of efficient algorithms for performing linear algebra computations.
- Linear algebra computations are at the core **many** algorithms. For example:
  - Linear regression uses matrix multiplication and inversion.
  - PageRank, the algorithm that spawned Google, was fundamentally calculating eigenvalues.
  - Many of the recommendation systems that power sites like Netflix, Amazon, Google Ads, etc. are based on matrix decomposition.

Element-wise Operations

- The most basic operations we have discussed are element-wise operations.

Element-wise Operations

- The most basic operations we have discussed are element-wise operations.
- Example: What is the complexity of multiplying an $n \times n$ matrix by a scalar?

Element-wise Operations

- The most basic operations we have discussed are element-wise operations.
- Example: What is the complexity of multiplying an $n \times n$ matrix by a scalar?

Element-wise Operations

- The most basic operations we have discussed are element-wise operations.
- Example: What is the complexity of multiplying an $n \times n$ matrix by a scalar?

```
Input: A, alpha
Output: B
for i in 1,...,n:
    for j in 1,...,n:
        B[i,j] = alpha*A[i,j]
```

Element-wise Operations

- The most basic operations we have discussed are element-wise operations.
- Example: What is the complexity of multiplying an $n \times n$ matrix by a scalar?

```
Input: A, alpha
Output: B
for i in 1,...,n:
    for j in 1,...,n:
        B[i,j] = alpha*A[i,j]
```

Scalar/matrix operations have complexity $\mathcal{O}(n^2)$ for an $n \times n$ matrix.

Element-wise Operations

- What about matrix addition?

Element-wise Operations

- What about matrix addition?

Element-wise Operations

- What about matrix addition?

```
Input: A, B
Output: C
for i in 1,...,n:
    for j in 1,...,n:
        C[i,j] = A[i,j] + B[i,j]
```

Element-wise Operations

- What about matrix addition?

```
Input: A, B
Output: C
for i in 1,...,n:
    for j in 1,...,n:
        C[i,j] = A[i,j] + B[i,j]
```

Element-wise matrix/matrix operations have complexity $\mathcal{O}(n^2)$ for two $n \times n$ matrices.

Inner Products

- A useful skill for evaluating numerical algorithms is translation between sums/products and loops.

Inner Products

- A useful skill for evaluating numerical algorithms is translation between sums/products and loops.
- Take for example the inner product of two length n vectors, x and y.

Inner Products

- A useful skill for evaluating numerical algorithms is translation between sums/products and loops.
- Take for example the inner product of two length n vectors, x and y.

Inner Products

- A useful skill for evaluating numerical algorithms is translation between sums/products and loops.
- Take for example the inner product of two length n vectors, x and y.

$$x^T y = \sum_i x_i y_i$$

Inner Products

- A useful skill for evaluating numerical algorithms is translation between sums/products and loops.
- Take for example the inner product of two length n vectors, x and y.

$$x^T y = \sum_i x_i y_i$$

```
Input: x, y
Output: out
out = 0
for i in 1,...,n:
    out += x[i]*y[i]
```

Inner Products

- A useful skill for evaluating numerical algorithms is translation between sums/products and loops.
- Take for example the inner product of two length n vectors, x and y.

$$x^T y = \sum_i x_i y_i$$

```
Input: x, y
Output: out
out = 0
for i in 1,...,n:
    out += x[i]*y[i]
```

A single sum/product can often be translated directly to a loop. This means that the inner product of two length n vectors has complexity $\mathcal{O}(n)$

Outer Products

Outer products, on the other hand, contain no sums. Instead, the output is an $n \times n$ matrix.

Outer Products

Outer products, on the other hand, contain no sums. Instead, the output is an $n \times n$ matrix.

$$A = xy^T \text{ s.t. } A_{ij} = x_i y_j$$

Outer Products

Outer products, on the other hand, contain no sums. Instead, the output is an $n \times n$ matrix.

$$A = xy^T \text{ s.t. } A_{ij} = x_i y_j$$

```
Input: x, y
Output: A
out = 0
for i in 1,...,n:
    for j in 1,...,n:
        A[i,j] = x[i]*y[j]
```

Outer Products

Outer products, on the other hand, contain no sums. Instead, the output is an $n \times n$ matrix.

$$A = xy^T \text{ s.t. } A_{ij} = x_i y_j$$

```
Input: x, y
Output: A
out = 0
for i in 1,...,n:
    for j in 1,...,n:
        A[i,j] = x[i]*y[j]
```

Outer products have complexity $\mathcal{O}(n^2)$.

Matrix Multiplication

Matrix multiplication can be decomposed into a bunch of vector products. Let A and B be $n \times n$ matrices, then

Matrix Multiplication

Matrix multiplication can be decomposed into a bunch of vector products. Let A and B be $n \times n$ matrices, then

$$C = AB \text{ s.t. } C_{ij} = A_{i:}B_{:j} = \sum_k A_{ik}B_{kj}$$

Matrix Multiplication

Matrix multiplication can be decomposed into a bunch of vector products. Let A and B be $n \times n$ matrices, then

$$C = AB \text{ s.t. } C_{ij} = A_{i:}B_{:j} = \sum_k A_{ik}B_{kj}$$

```
Input: A, B
Output: C
for i in 1,...,n:
    for j in 1,...,n:
        C[i,j] = 0
        for k in 1,...,n:
            C[i,j] += A[i,k]*B[k,j]
```

Matrix Multiplication

Matrix multiplication can be decomposed into a bunch of vector products. Let A and B be $n \times n$ matrices, then

$$C = AB \text{ s.t. } C_{ij} = A_{i:}B_{:j} = \sum_k A_{ik}B_{kj}$$

```
Input: A, B
Output: C
for i in 1,...,n:
    for j in 1,...,n:
        C[i,j] = 0
        for k in 1,...,n:
            C[i,j] += A[i,k]*B[k,j]
```

The naive method for matrix multiplication has complexity $\mathcal{O}(n^3)$.

Strassen's Algorithm

Modern linear algebra libraries actually use more advanced matrix multiplication algorithms based on **Strassen's Algorithm**.

Strassen's Algorithm

Modern linear algebra libraries actually use more advanced matrix multiplication algorithms based on **Strassen's Algorithm**.

Strassen's Algorithm: Basic Idea

- The naive matrix multiplication method requires 8 operations to multiply 2×2 matrices, however, Strassen found a way to do it in 7 operations at higher cost per operation.

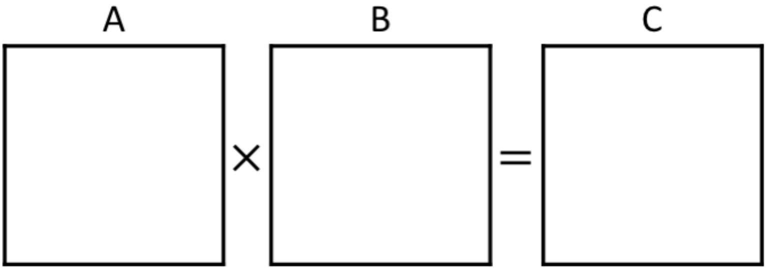
Strassen's Algorithm

Modern linear algebra libraries actually use more advanced matrix multiplication algorithms based on **Strassen's Algorithm**.

Strassen's Algorithm: Basic Idea

- The naive matrix multiplication method requires 8 operations to multiply 2×2 matrices, however, Strassen found a way to do it in 7 operations at higher cost per operation.
- Strassen's algorithm recursively divides a matrix into 2×2 matrices and applies Strassen's 2×2 method.

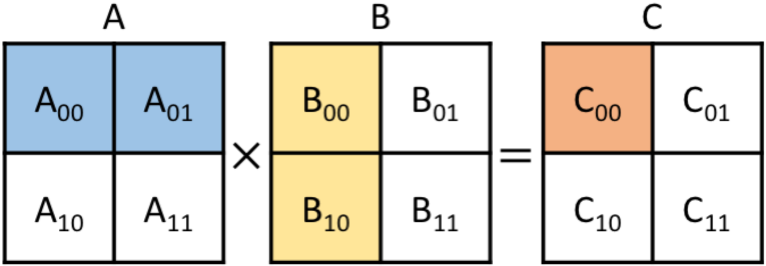
Strassen's Algorithm



Strassen's Algorithm

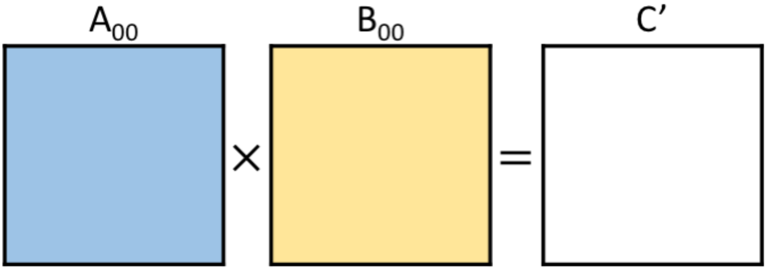
$$\overset{A}{\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}} \times \overset{B}{\begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}} = \overset{C}{\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}}$$

Strassen's Algorithm

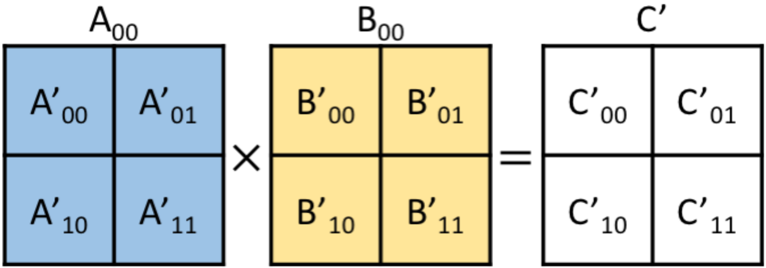


$$C_{00} = A_{00}B_{00} + A_{01}B_{10}$$

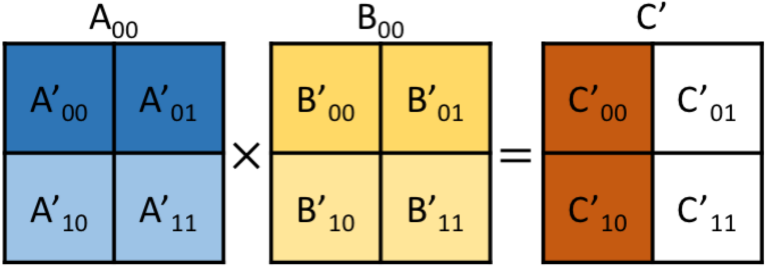
Strassen's Algorithm



Strassen's Algorithm



Strassen's Algorithm



$$C'_{00} = A'_{00}B'_{00} + A'_{01}B'_{10}$$

Strassen's Algorithm

- Strassen's algorithm reduces the complexity of matrix multiplication from $\mathcal{O}(n^3)$ to $\approx \mathcal{O}(n^2.807)$.

Strassen's Algorithm

- Strassen's algorithm reduces the complexity of matrix multiplication from $\mathcal{O}(n^3)$ to $\approx \mathcal{O}(n^2.807)$.
- Many extensions to Strassen's have further reduced the complexity. Numpy uses one of these (see Demo).

Strassen's Algorithm

- Strassen's algorithm reduces the complexity of matrix multiplication from $\mathcal{O}(n^3)$ to $\approx \mathcal{O}(n^2.807)$.
- Many extensions to Strassen's have further reduced the complexity. Numpy uses one of these (see Demo).
- Strassen's is typically faster for matrices of size 100×100 and above. Good implementations will switch between naive and Strassen's.

Demo

Demo