Divide and Conquer Algorithms

CS 4104: Data and Algorithm Analysis

Yoseph Berhanu Alebachew

May 11, 2025

Virginia Tech

Table of contents

Introduction

Divide and Conquer: Definition

- Divide and conquer refers to a class of algorithmic techniques.
- **Process:**
  - **Break:** Divide the input into several parts.
  - **Solve:** Solve the problem in each part recursively.
  - **Combine:** Combine the solutions to these subproblems into an overall solution.
- **Characteristics:**
  - **Simplicity:** Often a straightforward method.
  - **Power:** Can be a powerful technique for solving complex problems
- Will also become useful when we discuss other design techniques (e.g., Dynamic Programming)

Divide and Conquer: Runtime

- Involves solving a recurrence relation.
- Bounds the running time recursively.
- Analyze in terms of the running time on smaller instances.
- **Previous Lectures (Greedy Algorithms)**:
  - Brute Force Approach: Exponential running time.
  - Greedy Algorithm: Reduced running time to polynomial.
- **Divide and Conquer (most of the time)**:
  - Natural Brute-Force Algorithm: May already be polynomial time.
  - Strategy: Serves to reduce the running time to a lower polynomial.
- For example, the brute-force algorithm for finding the closest pair among n points in the plane would measure all $\Theta(n^2)$ distances, for a (polynomial) running time of $\Theta(n^2)$.
- Using divide and conquer will improve the running time to $O(nlogn)$.

The Sorting Problem

- Sorting is a common problem
- As a reminder it is the process of arranging elements in a specific order
- Common orders include numerical and lexicographical.
- Formal Problem Statements:
  - Input: A sequence of n numbers $a_1, a_2, \ldots, a_n$.
  - Output: A permutation $a'_1, a'_2, \ldots, a'_n$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- Basic algorithms such as Bubble, Insertion and Selection Sort have $O(n^2$
- Can we do better? Can we use divide and conquer approach

Mergesort: The Algorithm

Algorithm 1 Mergesort

```
procedure MERGESORT(A, left, right)
if left < right then
    mid =(left + right)/2
    MERGESORT(A, left, mid)
    MERGESORT(A, mid + 1, right)
    MERGE(A, left, mid, right)
end if
end procedure
```

Mergesort: The Algorithm

Algorithm 2 Mergesort - Merge Procedure

```
procedure MERGE(A, left, mid, right)
n1 = mid - left + 1
n2 = right - mid
create arrays L[1..n1] and R[1..n2]
for i = 1ton1 do
    L[i] = A[left + i − 1]
end for
for j = 1 to n2 do
    R[j] = A[mid + j]
end for
i = 1, j = 1
for k = left to right do
    if i <= n1 and (j > n2 or L[i] <= R[j]) then
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
    end if
end for
end procedure
```

Mergesort: Example

- Initial Array:

38	27	43	3	9	82	10

Mergesort: Example

- Initial Array:

38	27	43	3	9	82	10

- Split Step 1:

38	27	43	3

9	82	10

Mergesort: Example

- Initial Array:

38	27	43	3	9	82	10

- Split Step 1:

38	27	43	3

9	82	10

- Split Step 2:

38	27

43	3

9	82

10

Mergesort: Example

- Initial Array:

38	27	43	3	9	82	10

- Split Step 1:

38	27	43	3

9	82	10

- Split Step 2:

38	27

43	3

9	82

10

- Split Step 3:

38

27

43

3

9

82

10

Mergesort: Example

- Merge Step 1:

38	27	43	3	9	82	10

Mergesort: Example

- Merge Step 1:

38	27	43	3	9	82	10

- Merge Step 2:

27	38	3	43	9	82	10

Mergesort: Example

- Merge Step 1:

38	27	43	3	9	82	10

- Merge Step 2:

27	38	3	43	9	82	10

- Merge Step 3:

3	27	38	43	9	10	82

Mergesort: Example

- Merge Step 1:

38	27	43	3	9	82	10

- Merge Step 2:

27	38	3	43	9	82	10

- Merge Step 3:

3	27	38	43	9	10	82

- Sorted Array:

3	9	10	27	38	43	82

Mergesort: Correctness

- Correctness:
  - Each step of dividing and merging ensures the subarrays are sorted.
  - Final merge produces a completely sorted array.
  - Merging two sorted lists by picking the smallest item from the head of each list at a time ensures the end result is sorted
- Proof by induction:
  - Base Case: Single element arrays are trivially sorted.
  - Inductive Step: Merging two **sorted** arrays in picking the smallest item from the two heads **maintains** order.
  - Termination: The algorithm terminates when all elements are merged back together

Mergesort: Runtime

- Recurrence Relation: $T(n) = 2T(n/2) + O(n)$
  - Divide the array into two halves.
  - Recursively sort each half.
  - Merge the halves in linear time.

Mergesort: Runtime

- Recurrence Relation: $T(n) = 2T(n/2) + O(n)$
  - Divide the array into two halves.
  - Recursively sort each half.
  - Merge the halves in linear time. Poll 1

Mergesort: Runtime

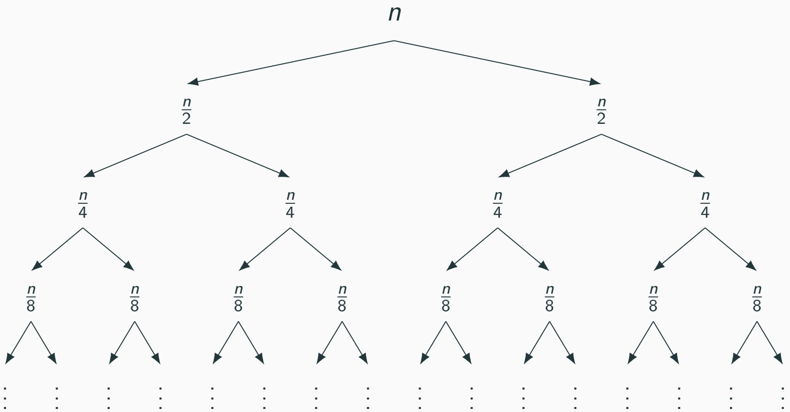
- Recurrence Relation: $T(n) = 2T(n/2) + O(n)$
  - Divide the array into two halves.
  - Recursively sort each half.
  - Merge the halves in linear time. Poll 1
- Solution:
  - Using the Master Theorem (Will discuss later):
    - $a = 2, b = 2, f(n) = O(n)$
    - $T(n) = O(nlogn)$
- Time Complexity:
  - Best, Average, and Worst Case: $O(nlogn)$

Solving Recurrences

- How can we solve for $T(n) = 2T(n/2) + n$?
  - How to find the asymptotic bound
- Three methods:
  - Recursive Tree
    - Converts the recurrence into a tree.
    - Draw the recursion tree.
    - Sum the costs of all levels.
  - Substation Method
    - Substitute inside the equation
  - Master theorem
    - Check cases and use the rule

Recursive Tree: Example 1

- **Example**: Solve $T(n) = 2T(n/2) + n$.



Question: What patterns can we observe from the tree structure?

Recursive Tree: Example 1

- Pattern for $T(n) = 2T(n/2) + n$.
  - Level 0: $2(n) = n = 2^0(n/2^0)$
  - Level 1: $2(n/2) = n = 2^1(n/2^1)$.
  - Level 2: $4(n/4) = n = 2^2(n/2^2)$.
  - Level 3: $8(n/8) = n = 2^3(n/2^3)$.
  - Level i: $2^i(n/2^i)$.
- Sum the cost at each level
- Lets say the height of the tree is h
  - Then $T(n) = h * n$

Questions:

- What will be the maximum value of i ?
  - What will be the height the tree ?
  - What will be the cost of a node at the leaf ?
    - Poll 2

Recursive Tree: Example 1

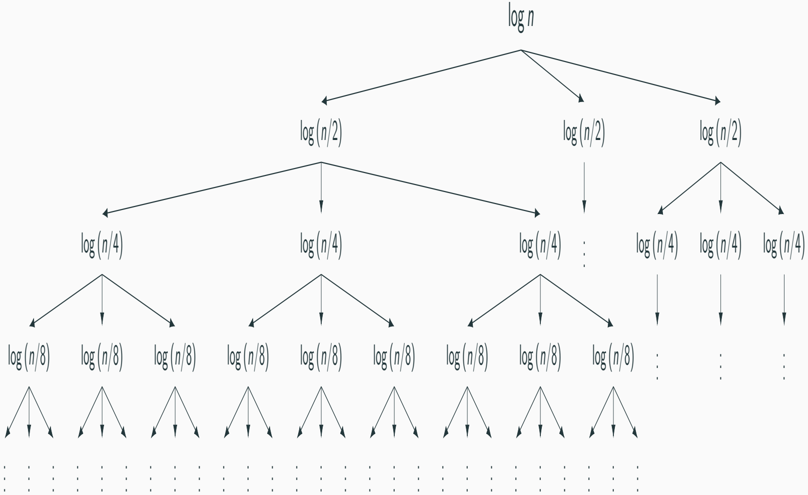
- Pattern for $T(n) = 2T(n/2) + n$.
  - Level 0: $2(n) = n = 2^0(n/2^0)$
  - Level 1: $2(n/2) = n = 2^1(n/2^1)$.
  - Level 2: $4(n/4) = n = 2^2(n/2^2)$.
  - Level 3: $8(n/8) = n = 2^3(n/2^3)$.
  - Level i: $2^i(n/2^i)$.
- Sum the cost at each level
- Lets say the height of the tree is h
  - Then $T(n) = h * n$

Questions:

- What will be the maximum value of i ?
  - What will be the height the tree ?
  - What will be the cost of a node at the leaf ?
    - Poll 2 The value of (n/x) becomes 1
- How many leaves will the tree have

Recursive Tree: Example 2

- **Example**: Solve $T(n) = 3T(n/2) + \log n$.



Recursive Tree: Example 2

- Pattern for $T(n) = 3T(n/2) + \log n$.
  - Level 0: $\log n$.
  - Level 1: $3 \cdot \log(n/2)$.
  - Level 2: $3^2 \cdot \log(n/4)$.
  - Level 3: $3^3 \cdot \log(n/8)$.
  - Level i: $3^i \cdot \log(n/2^i)$.
- Sum the cost at each level.
  - Then $T(n)$ is the sum of the costs of all levels.

Questions:

- What will be the maximum value of i?
  - What will be the height of the tree?
  - What will be the cost of the node at the leaf?
- How many leaves will the tree have?

Recursive Tree: Example 2

- Pattern for $T(n) = 3T(n/2) + \log n$.
  - Level 0: $\log n$.
  - Level 1: $3 \cdot \log(n/2)$.
  - Level 2: $3^2 \cdot \log(n/4)$.
  - Level 3: $3^3 \cdot \log(n/8)$.
  - Level i: $3^i \cdot \log(n/2^i)$.
- Sum the cost at each level.
  - Then $T(n)$ is the sum of the costs of all levels.

Questions:

- What will be the maximum value of i?
  - What will be the height of the tree?
  - What will be the cost of the node at the leaf?
- How many leaves will the tree have?

Poll 3

- Sum up to $\log n$ levels: $\sum_{i=0}^{\log n} 3^i \log(n/2^i)$.
- Simplify the series to find the overall complexity.

Recursive Tree: Example 2

- Given the sum: $\sum_{i=0}^{\log n} 3^i \log\left(\frac{n}{2^i}\right)$.
  - Rewrite the term inside the summation:
  $$3^i \log\left(\frac{n}{2^i}\right) = 3^i \log n - 3^i i \log 2$$
  - Split the sum:
  $$\sum_{i=0}^{\log n} 3^i \log n - \sum_{i=0}^{\log n} 3^i i \log 2$$
  - Analyze the first sum:
  $$\sum_{i=0}^{\log n} 3^i \log n = \log n \sum_{i=0}^{\log n} 3^i$$
  - The geometric series sum:
  $$\sum_{i=0}^{\log n} 3^i = \frac{3^{\log n+1} - 1}{2} \approx \frac{3n^{\log 3}}{2} = O(n^{\log 3})$$
  - Therefore:
  $$\log n \cdot O(n^{\log 3}) = O(n^{\log 3} \log n)$$

Recursive Tree: Example 2

- Analyze the second sum: $\sum_{i=0}^{\log n} 3^i i \log 2 = \log 2 \sum_{i=0}^{\log n} 3^i i$
  - The sum $\sum_{i=0}^{\log n} 3^i i$ is dominated by its largest term when $i \approx \log n$:

$$\sum_{i=0}^{\log n} 3^i i \approx (\log n) 3^{\log n} = (\log n) n^{\log 3}$$

  - Therefore:

$$\log 2 \cdot O((\log n) n^{\log 3}) = O((\log n) n^{\log 3})$$

- Combining both sums:

$$\sum_{i=0}^{\log n} 3^i \log\left(\frac{n}{2^i}\right) = O(n^{\log 3} \log n) + O((\log n) n^{\log 3})$$

- The first term dominant so our overall complexity is: $O((\log n) n^{\log 3})$

Substitution Method: Introduction

- The substitution method is used to find asymptotic bounds on recurrence relations.
- Steps:
  - Guess the form of the solution.
  - Use mathematical induction to find constants and show the solution fits.
  - You can use recursion tree to guess the solution

Question:

- Why is guessing the form of the solution important in the substitution method?
- What happens if our initial guess for $T(n)$ is incorrect?

Poll 4

Substitution Method: Introduction

- The substitution method is used to find asymptotic bounds on recurrence relations.
- Steps:
  - Guess the form of the solution.
  - Use mathematical induction to find constants and show the solution fits.
  - You can use recursion tree to guess the solution

Question:

- Why is guessing the form of the solution important in the substitution method?
- What happens if our initial guess for $T(n)$ is incorrect?

Poll 4

Substitution Method: Example 1

- Solve $T(n) = 2T(n/2) + n$.
  - Guess: $T(n) = O(n \log n)$.
  - Task: Show $T(n) \leq c(n \log n)$.
  - Base case: for $T(1)$, can find a constant c such that $T(1) \leq c$.
  - Inductive step: assume $T(\frac{k}{2}) \leq c\frac{k}{2} \log \frac{k}{2}$ for a $\frac{k}{2} < n$.
  - Now show for k that $T(k) \leq c(k \log k)$

$$T(k) = 2T(\frac{k}{2}) + k$$

  - Replace $T(\frac{k}{2})$ from our assumption

$$T(k) \leq 2c\frac{k}{2} \log \frac{k}{2} + k$$

$$T(k) \leq ck \log k - ck \log 2 + k$$

  - Since our base for log is 2 $\log 2 = 1$

$$T(k) \leq ck \log k - ck + k$$

$$T(k) \leq ck \log k - (c-1)k$$

  - Notice $(c-1)k$ is a positive number, therefore

$$T(k) \leq ck \log k \Rightarrow T(k) = O(k \log k)$$

Substitution Method: Example 2

- Solve $T(n) = 3T(n/3) + n$.
  - Guess: $T(n) = O(n \log n)$.
  - Task: Show $T(n) \leq c(n \log n)$.
  - Base case: for $T(1)$, we can find a constant c such that $T(1) \leq c$.
  - Inductive step: assume $T(\frac{k}{3}) \leq c\frac{k}{3} \log \frac{k}{3}$ for a $\frac{k}{3} < n$.
  - Now show for k that $T(k) \leq c(k \log k)$

$$T(k) = 3T(\frac{k}{3}) + k$$

  - Replace $T(\frac{k}{3})$ from our assumption

$$T(k) \leq 3c\frac{k}{3} \log \frac{k}{3} + k$$

$$T(k) \leq ck \log k - ck \log 3 + k$$

  - Combine terms:

$$T(k) \leq ck \log k - (c \log 3 - 1)k$$

  - Notice $(c \log 3 - 1)k$ is a positive number, therefore

$$T(k) \leq ck \log k \Rightarrow T(k) = O(k \log k)$$

Master Theorem: Introduction

- The Master Theorem provides a straightforward way to solve recurrences of the form:

$$T(n) = aT(n/b) + f(n)$$

- It applies to divide-and-conquer algorithms where the problem is divided into a subproblems, each of size n/b, and $f(n)$ represents the cost outside the recursive calls.
- Advantages:
  - Provides a quick method to determine the time complexity of recursive algorithms.
  - Helps in identifying the dominant term in the recurrence.
- Limitations:
  - Not applicable to all types of recurrences, especially those with non-polynomial $f(n)$.
  - Assumes that the recurrence divides the problem into equal-sized subproblems.

Master Theorem

- For $a \geq 1 and b > 1$ in a recursion of the form

$$T(n) = aT(n/b) + f(n)$$

- Master defines three cases:
  - If $f(n) = O(n^{\log_b a})$, then $T(n) = O(n^{\log_b a})$.
  - If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = O(n^{\log_b a} \log n)$.
  - If $f(n) = \Omega(n^{\log_b a})$, and if $af(n/b) \leq kf(n)$ for some $k < 1$ and sufficiently large n, then $T(n) = O(f(n))$.
- Important considerations:
  - The function $f(n)$ must be *polynomially* bounded.
  - The Master Theorem does not apply if $f(n)$ is not in the form of $O(n^c)$.
  - More generally the function $f(n)$ should be positive and asymptotically non-decreasing. .

Master Theorem: Examples

- Solve $T(n) = 2T(n/2) + n$.
  - Here, $a = 2$, $b = 2$, and $f(n) = n$.
  - $\log_b a = \log_2 2 = 1$
  - Compare $f(n) = n$ with $n^{\log_b a} = n^1$
  - **Case 2**: $f(n) = O(n^{\log_b a})$
  - Therefore, $T(n) = O(n \log n)$.
- Solve $T(n) = 3T(n/4) + \log n$.
  - Here, $a = 3$, $b = 4$, and $f(n) = \log n$.
  - $\log_b a = \log_4 3 \approx 0.792$
  - Compare $f(n) = \log n$ with $n^{\log_b a} = n^{0.792}$
  - **Case 1**: $f(n) = O(n^{\log_b a})$
  - Therefore, $T(n) = O(n^{\log_b a}) = O(n^{0.792})$.

Master Theorem: Examples

- Solve $T(n) = 4T(n/2) + n^2$.
  - Here, $a = 4$, $b = 2$, and $f(n) = n^2$.
  - $\log_b a = \log_2 4 = 2$
  - Compare $f(n) = n^2$ with $n^{\log_b a} = n^2$
  - **Case 2**: $f(n) = \Theta(n^{\log_b a})$
  - Therefore, $T(n) = O(n^{\log_b a} \log n) = O(n^2 \log n)$.

- Solve $T(n) = 3T(n/2) + n^3$.
  - Here, $a = 3$, $b = 2$, and $f(n) = n^3$.
  - $\log_b a = \log_2 3 \approx 1.585$
  - Compare $f(n) = n^3$ with $n^{\log_b a} = n^{1.585}$
  - **Case 3**: $f(n) = \Omega(n^{\log_b a})$
  - Also, $af(n/b) \leq kf(n)$ for some $k < 1$
  - Therefore, $T(n) = O(f(n)) = O(n^3)$.

Example Problems

Counting Inversions: The Problem

- An inversion in an array $A[1 \dots n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$.
- The problem is to count the number of inversions in the array.
- Inversions indicate how far the array is from being sorted.
- Consider the array $A = [2, 4, 1, 3, 5]$
- The inversions are:
  - $(2, 1)$
  - $(4, 1)$
  - $(4, 3)$
- Thus, the total number of inversions is 3.
- How many inversion does the array $A = [2, 6, 4, 3, 8, 11]$?
  - Answer: $3 \Rightarrow (6, 4), (6, 3), (4, 3)$

Question: How many inversions does a completely sorted array have?

Counting Inversions: Brute-force solution

- A simple approach is to use a nested loop to count all inversions.

Algorithm 3 Brute-force Inversion Count

```
count = 0
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if A[i] > A[j] then
            count = count + 1
        end if
    end for
end for

return count
```

Question: What is the time complexity of this brute-force solution? Poll 5

Counting Inversions: Brute-force solution

- A simple approach is to use a nested loop to count all inversions.

Algorithm 4 Brute-force Inversion Count

```
count = 0
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if A[i] > A[j] then
            count = count + 1
        end if
    end for
end for

return count
```

Question: What is the time complexity of this brute-force solution? Poll 5

Counting Inversions: Divide and Conquer algorithm

- We can use a divide-and-conquer approach, similar to merge sort, to count inversions more efficiently.
- The idea is to:
  - Divide the array into two halves.
  - Count the inversions in each half.
  - Count the inversions that cross the two halves.
- This approach can reduce the time complexity significantly.

Question: What if we use merge sort as is and count the number of time we picked the item from the right halve on the merge step ? Poll 6

Counting Inversions: Divide and Conquer algorithm

- We can use a divide-and-conquer approach, similar to merge sort, to count inversions more efficiently.
- The idea is to:
    - Divide the array into two halves.
    - Count the inversions in each half.
    - Count the inversions that cross the two halves.
- This approach can reduce the time complexity significantly.

Question: What if we use merge sort as is and count the number of time we picked the item from the right halve on the merge step ? Poll 6

Counting Inversions: Pseudo-code

Algorithm 5 Counting Inversions

```
function COUNTINGINVERSIONS(arr, n)
    return mergeSort(arr, 0, n-1)
function mergeSortCountInversion(arr, left, right)
    inv = 0
if left ¡ right then
        mid = (left + right) // 2
        inv += mergeSortCountInversion(arr, left, mid)
        inv += mergeSortCountInversion(arr, mid + 1, right)
        inv += MERGE(arr, left, mid, right)
end if
    return inv
```

Counting Inversions: Pseudo-code

Algorithm 6 Merge Initialization

```
function MERGE(arr, left, mid, right)
  i = left, j = mid + 1, inv = 0
while i ≤ mid and j ≤ right do
    if arr[i] ≤ arr[j] then
          i = i + 1
    else
          temp = arr[j]
          shift arr[i:j-1] right
          arr[i] = temp
          inv += (mid - i + 1)
          i = i + 1
          mid = mid + 1
          j = j + 1
    end if
end while
while j ≤ right do
        temp = arr[j]
        shift arr[i:j-1] right
        arr[i] = temp
        inv_count += (mid - i + 1)
        i = i + 1
        mid = mid + 1
        j = j + 1
end while
  return inv_count
```

Counting Inversions: Example

- Consider the array $A = [2, 4, 1, 3, 5]$

Counting Inversions: Example

- Consider the array $A = [2, 4, 1, 3, 5]$
- Initial array:

2	4	1	3	5

Counting Inversions: Example

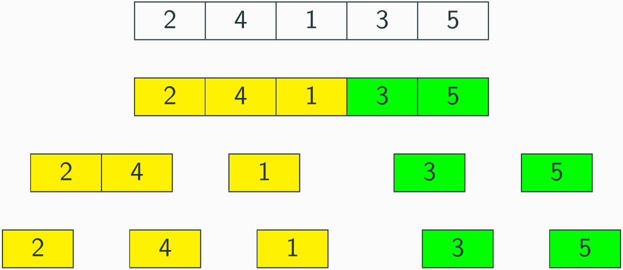
- Consider the array $A = [2, 4, 1, 3, 5]$
- Initial array:

2	4	1	3	5

2	4	1	3	5

Counting Inversions: Example

- Consider the array $A = [2, 4, 1, 3, 5]$
- Initial array:



Counting Inversions: Example

Inversions: 0

- Merge and Count Inversions:

2	**4**	1	3	5

Counting Inversions: Example

Inversions: 0

- Merge and Count Inversions:

2	4	1	3	5

Counting Inversions: Example

Inversions:	2

- Merge and Count Inversions:

2	4	1	3	5

Counting Inversions: Example

Inversions: 2

- Merge and Count Inversions:

1	2	4	3	5

Counting Inversions: Example

Inversions:	2

- Merge and Count Inversions:

1	2	4	**3**	**5**

Counting Inversions: Example

Inversions:	2

- Merge and Count Inversions:

1	2	4

3	5

Counting Inversions: Example

Inversions:	2

- Merge and Count Inversions:

1	2	**4**		3	5

Counting Inversions: Example

Inversions: 2

- Merge and Count Inversions:

1	**2**	4		**3**	5

Counting Inversions: Example

Inversions:	2

- Merge and Count Inversions:

1	2	**4**

3	5

Counting Inversions: Example

Inversions:	3

- Merge and Count Inversions:

1	2	**4**

3	**5**

Counting Inversions: Example

Inversions:	3

- Merge and Count Inversions:

1	2	3	4	5

Counting Inversions: Correctness

- The divide-and-conquer algorithm correctly counts inversions because:
  - It divides the array into two halves and counts inversions in each half.
  - It counts the inversions that cross the two halves during the merge step.
- The correctness follows from the correctness of merge sort, where each element is compared and merged correctly.

Counting Inversions: Running Time

- The time complexity of the divide-and-conquer algorithm can be analyzed as follows:
  - The recurrence relation is $T(n) = 2T(n/2) + O(n)$.
  - This is the same as merge sort, which solves to $T(n) = O(n \log n)$.
- Therefore, the running time of the inversion counting algorithm is $O(n \log n)$.

Question: How does the time complexity of the divide-and-conquer approach compare to the brute-force solution?

Counting Inversions: Applications

- **Sorting and Order Statistics:**
  - Counting inversions can measure how far an array is from being sorted.
  - Useful in evaluating and improving sorting algorithms.
- **Genome Analysis:**
  - In bioinformatics, counting inversions helps in genome rearrangement problems.
  - Used to study the evolutionary distance between species by comparing genome sequences.
- **Rank Correlation:**
  - Spearman's footrule and Kendall's tau distance between two rankings can be computed using inversion count.
  - Important in statistics for comparing ranked lists.
- **Network Theory:**
  - Inversions can help analyze network reliability and failure rates.
  - Used to study the robustness of network topologies.
- **In Economics:**
  - Counting inversions can model and analyze discrepancies in economic indicators.

Closest Pair of Points: Problem Definition

- Given a set of points in a plane, find the pair of points with the minimum Euclidean distance between them.
- **Input:** A set of points $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$.
- **Output:** The pair of points (p_1, p_2) such that the distance $d(p_1, p_2)$ is minimized.
- **Applications:**
  - Computational geometry problems.
  - Pattern recognition and clustering.
  - Network design and layout optimization.

Closest Pair of Points: Brute-force solution

- **Algorithm:**
  - Initialize *min_dist* to infinity.
  - For each pair of points (p_i, p_j) in the set P:
    - Compute the distance $d(p_i, p_j)$.
    - If $d(p_i, p_j) < min\_dist$, update *min_dist* and the closest pair.
- **Time Complexity:** $O(n^2)$.
- This solution checks all possible pairs, making it inefficient for large datasets.

Closest Pair of Points: Divide and Conquer algorithm

- The divide-and-conquer approach improves efficiency.
- **Steps:**
  - Sort the points by their x-coordinates.
  - Recursively find the closest pair in the left and right halves.
  - Find the closest pair that straddles the dividing line.
  - Combine these results to find the overall closest pair.
- **Time Complexity:** $O(n \log n)$.

Closest Pair of Points: Pseudo-code

Algorithm 7 Closest Pair of Points

function ClosestPair(P)
    Sort P by x-coordinates
    return ClosestPairRec(P)

function ClosestPairRec(P)
if length(P) $\leq$ 3 **then**
        return BruteForce(P)
end if
    mid = length(P) / 2
    $L = P[1 \ldots mid]$
    $R = P[mid + 1 \ldots end]$
    $(p_1, q_1) =$ ClosestPairRec(L)
    $(p_2, q_2) =$ ClosestPairRec(R)
    $\delta = \min(d(p_1, q_1), d(p_2, q_2))$
    $M =$ points in P within δ of the dividing line
    $(p_3, q_3) =$ ClosestSplitPair(M, δ)
    return the pair with the smallest distance among (p_1, q_1), (p_2, q_2), and (p_3, q_3)

Closest Pair of Points: Closest Split Pair Function

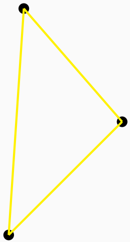
```
function ClosestSplitPair(M, δ)
   Sort M by y-coordinates
   best = δ
   best_pair = (nil, nil)
for i = 1 to length(M) do
   for j = i + 1 to min(i + 7, length(M)) do
      if d(M[i], M[j]) < best then
            best = d(M[i], M[j])
            best_pair = (M[i], M[j])
      end if
   end for
end for
   return best_pair
```

Closest Pair of Points: Visual Example - Base case

If there are less than 3 points
Compare the distance using the brute-force algorithm

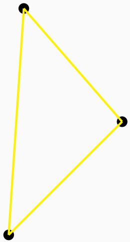
Closest Pair of Points: Visual Example - Base case

If there are less than 3 points
Compare the distance using the brute-force algorithm



Closest Pair of Points: Visual Example - Base case

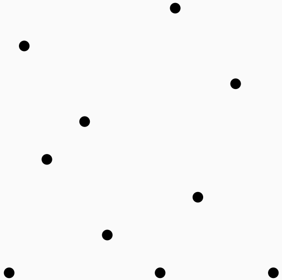
If there are less than 3 points
Compare the distance using the brute-force algorithm



This has a constant time complexity (three comparison)

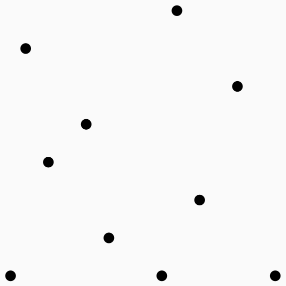
Closest Pair of Points: Visual Explanation

Input



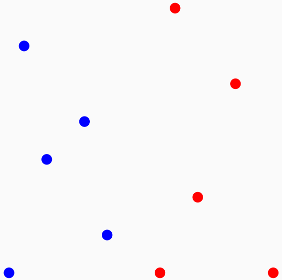
Closest Pair of Points: Visual Explanation

Sort the points based on their x value

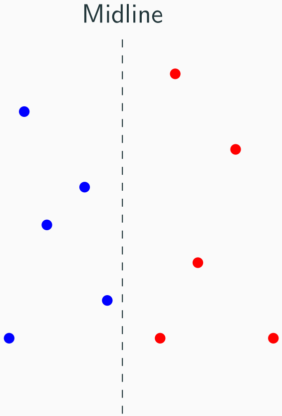


Closest Pair of Points: Visual Explanation

Split them in two halves

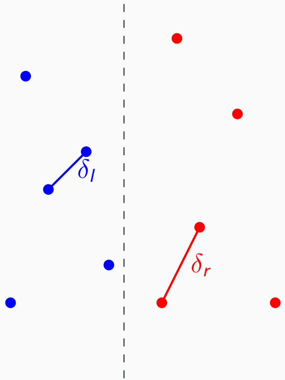


Closest Pair of Points: Visual Explanation



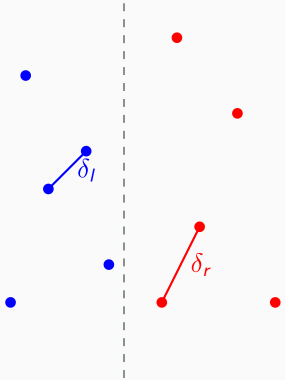
Closest Pair of Points: Visual Explanation

Find closest pairs in each half (done through a recursive call)



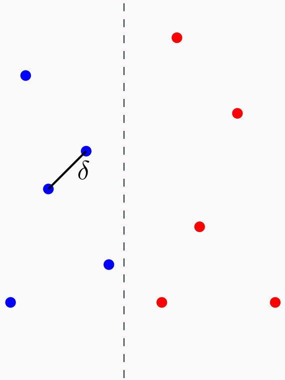
Closest Pair of Points: Visual Explanation

Compare δ_r and δ_l and take the smallest



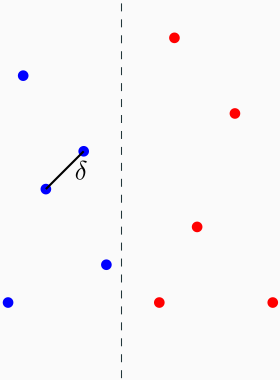
Closest Pair of Points: Visual Explanation

Let say δ_l is the smallest and we take that as δ



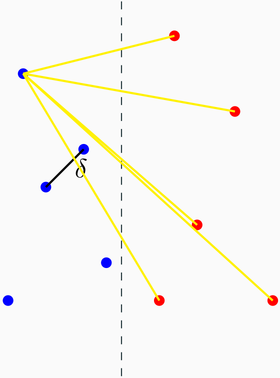
Closest Pair of Points: Visual Explanation

Check if there is a pair that has a smaller distance than δ that has a point in each of the halves



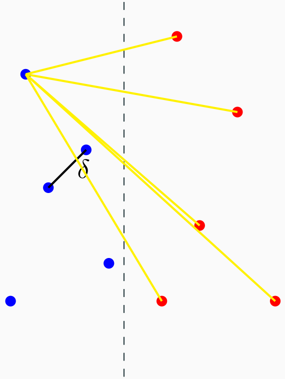
Closest Pair of Points: Visual Explanation

The naive way to do this is, to pair every point in one half
with all points in the other and check the distance is lower than δ



Closest Pair of Points: Visual Explanation

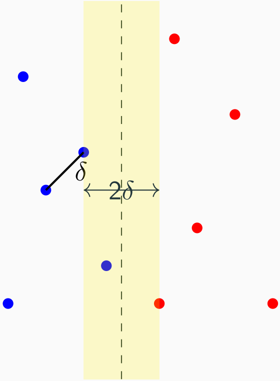
The naive way to do this is, to pair every point in one half
with all points in the other and check the distance is lower than δ



This won't be any better than the brute-force algorithm

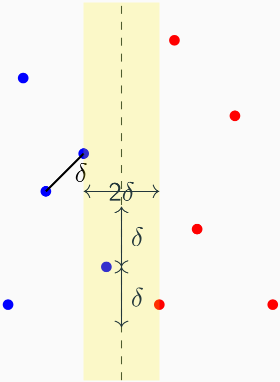
Closest Pair of Points: Visual Explanation

Notice we really don't need to check for points that are more than δ away from the midpoint (x-value)



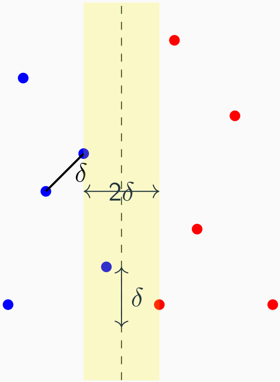
Closest Pair of Points: Visual Explanation

Even within this band, for a point we only need to check it's pairing to points that are a maximum of δ away in their y-value



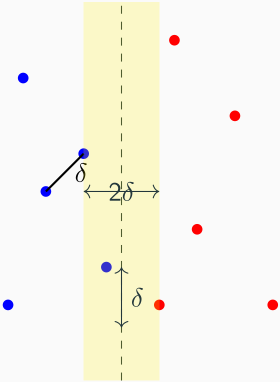
Closest Pair of Points: Visual Explanation

If we sort the points within this band by their y-value, for each point, we only need to check the points within δ distance in one direction

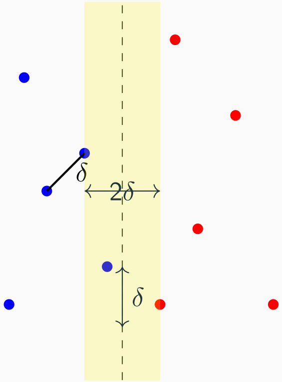


Closest Pair of Points: Visual Explanation

How will this reduce the number of points we need tp check ?



Closest Pair of Points: Visual Explanation



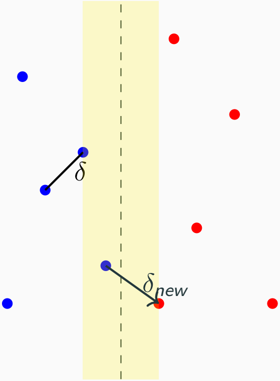
Claim: with these constraints, we only need to check a maximum of 6 points for each point in the band?

This makes comparing crossing points a constant time operation, per point

Meaning the upper bound of this operation will be $O(n)$

Closest Pair of Points: Visual Explanation

If we find a smaller distance, this new pair will be our closest pair of points at this level of the recursion



Closest Pair of Points: Takeaway

- How did the divide and conquer improve the runtime?
- Notice how the actual check is still a brute force method
- What changed is the candidates we need to consider at each step
- How did the approach help us reduce the brute force checks we make ?
    - By having the smallest distance in both halves, it gave us a cut-off point to remove most of the point from consideration

Closest Pair of Points: Correctness

- The divide-and-conquer algorithm correctly finds the closest pair of points.
- The correctness follows from:
  - The correctness of the recursive calls to find the closest pairs in the left and right halves.
  - The correctness of the merge step to find the closest split pair.
  - Ensuring that all potential closest pairs are considered.

Question: Why does the merge step only consider points within δ of the dividing line?

Closest Pair of Points: Running Time

- The time complexity of the divide-and-conquer algorithm can be analyzed as follows:
  - Sorting the points by x-coordinates takes $O(n \log n)$.
  - The recursive calls each handle half the points, leading to $2T(n/2)$.
  - The merge step takes $O(n)$ time.
- The recurrence relation is $T(n) = 2T(n/2) + O(n)$.
- Solving this using the Master Theorem gives $T(n) = O(n \log n)$.
- Therefore, the running time of the closest pair of points algorithm is $O(n \log n)$.

Question: How does the time complexity of the divide-and-conquer approach compare to the brute-force solution?

Closest Pair of Points: Applications

- **Computational Geometry:**
  - Widely used in geometric computations and computer graphics.
- **Astronomy:**
  - Finding the closest stars or celestial objects in space.
- **Geographical Information Systems (GIS):**
  - Finding the closest facilities (e.g., hospitals, schools) to a given location.
- **Networking:**
  - Optimizing the layout of network nodes to minimize latency.
- **Clustering:**
  - Used as a subroutine in clustering algorithms to group points based on proximity.

Maximum Sub-array: The Problem

- Given an array of integers, find the contiguous sub-array (containing at least one number) which has the largest sum.
- Example: For the array $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$, the contiguous sub-array with the largest sum is $[4, -1, 2, 1]$ with sum 6.

-2	1	-3	4	-1	2	1	-5	4

- The sub-array $[4, -1, 2, 1]$ has the largest sum, which is 6.

Maximum Sub-array: Brute-force Approach

- The naive approach involves checking all possible sub-array.
  - Initialize a variable *max_sum* to negative infinity.
  - Iterate through each sub-array and calculate its sum.
  - Update *max_sum* if the current sub-array sum is greater.

Algorithm 8 Brute-force Maximum Sub-array

Require: Array *arr* of length n

```
max_sum ← −∞
for i ← 0 to n − 1 do
    for j ← i to n − 1 do
        current_sum ← 0
        for k ← i to j do
            current_sum ← current_sum + arr[k]
        end for
        max_sum ← max(max_sum, current_sum)
    end for
end for
return max_sum
```

- **Time Complexity:** $O(n^3)$, where n is the length of the array.

Maximum Sub-array: Divide and Conquer Approach

- The Divide and Conquer approach splits the array into two halves and recursively finds the maximum sub-array sum.
- **Algorithm:**
  - Divide the array into two halves.
  - Recursively find the maximum sub-array sum in the left half and right half.
  - Find the maximum sub-array sum that crosses the midpoint.
  - Return the maximum of the three sums.

Maximum Sub-array: Pseudo-code

Algorithm 9 Divide and Conquer Maximum Sub-array

Require: Array *arr* of length n

Ensure: Maximum sub-array sum

if $n = 1$ **then**
    return $arr[0]$
end if
$mid \leftarrow \lfloor n/2 \rfloor$
$left\_max \leftarrow \text{max\_subarray}(arr, 0, mid - 1)$
$right\_max \leftarrow \text{max\_subarray}(arr, mid, n - 1)$
$cross\_max \leftarrow \text{max\_crossing\_subarray}(arr, 0, mid, n - 1)$
return $\max(left\_max, right\_max, cross\_max)$

Maximum Sub-array: Max Crossing Sub-array Pseudo-code

Algorithm 10 Max Crossing Sub-array

Require: Array *arr*, indices *low*, *mid*, *high*

Ensure: Maximum sub-array sum that crosses the midpoint

```
left_sum ← −∞
sum ← 0
for i ← mid to low step −1 do
    sum ← sum + arr[i]
    if sum > left_sum then
        left_sum ← sum
    end if
end for
right_sum ← −∞
sum ← 0
for j ← mid + 1 to high do
    sum ← sum + arr[j]
    if sum > right_sum then
        right_sum ← sum
    end if
end for
return left_sum + right_sum
```

Maximum Sub-array: Example

-2	1	-3	4	-1	2	1	-5	4

Maximum Sub-array: Example

-2	1	-3	4	-1	2	1	-5	4

-2	1	-3	4

-1	2	1	-5	4

Maximum Sub-array: Example

-2	1	-3	4	-1	2	1	-5	4

-2	1	-3	4

Right Sum: $-\infty$

Sum: $-\infty$

-1	2	1	-5	4

Left Sum: $-\infty$

Sum: $-\infty$

Maximum Sub-array: Example

-2	1	-3	4	-1	2	1	-5	4

-2	1	-3	4

Right Sum: 4

Sum: 4

-1	2	1	-5	4

Left Sum: -1

Sum: -1

Maximum Sub-array: Example

-2	1	-3	4	-1	2	1	-5	4

-2	1	-3	4

Right Sum: 4

Sum: 1

-1	2	1	-5	4

Left Sum: 1

Sum: 1

Maximum Sub-array: Example

-2	1	-3	4	-1	2	1	-5	4

-2	1	-3	4

Right Sum: 4

Sum: 2

-1	2	1	-5	4

Left Sum: 2

Sum: 2

Maximum Sub-array: Example

-2	1	-3	4	-1	2	1	-5	4

-2	1	-3	4

Right Sum: 4

Sum: 0

-1	2	1	-5	4

Left Sum: 2

Sum: -3

Maximum Sub-array: Example

-2	1	-3	4	-1	2	1	-5	4

-2	1	-3	4

Right Sum: 4

Sum: 0

-1	2	1	-5	4

Left Sum: 2

Sum: 1

Maximum Sub-array: Example

-2	1	-3	4	-1	2	1	-5	4

-2	1	-3	4

-1	2	1	-5	4

Max Crossing: $4 + 2 = 6$

Maximum Sub-array: Example

-2	1	-3	4	-1	2	1	-5	4

-2	1	-3	4

Max Left: 4

-1	2	1	-5	4

Max Right: 4

Max Crossing: $4 + 2 = 6$

Maximum Sub-array: Example

-2	1	-3	4	-1	2	1	-5	4

-2	1	-3	4

Max Left: 4

-1	2	1	-5	4

Max Right: 4

Max Crossing: $4 + 2 = 6$

Max Sub-array : 6

Maximum Subarray: Correctness

- The correctness of the Divide and Conquer approach follows from:
    - The maximum subarray sum must be in the left half, the right half, or cross the midpoint.
    - The algorithm correctly identifies the maximum sum in each of these cases.
    - By combining the results, the approach guarantees the correct maximum subarray sum.

Question: Why does the algorithm consider the cross sum?

Maximum Subarray: Kadane's Algorithm

- Kadane's Algorithm provides an efficient way to solve the Maximum Subarray problem.
- **Algorithm:**
  - Initialize two variables: *max_so_far* to negative infinity and *max_ending_here* to 0.
  - Iterate through the array.
  - At each element, add the element to *max_ending_here*.
  - If *max_ending_here* is greater than *max_so_far*, update *max_so_far*.
  - If *max_ending_here* is less than 0, reset it to 0.
- **Time Complexity:** $O(n)$, where n is the length of the array.

Maximum Subarray: Kadane's Algorithm Pseudo-code

Algorithm 11 Kadane's Algorithm

Require: Array *arr* of length n

Ensure: Maximum subarray sum

```
max_so_far ← −∞
max_ending_here ← 0
for i ← 0 to n − 1 do
    max_ending_here ← max_ending_here + arr[i]
    if max_so_far < max_ending_here then
        max_so_far ← max_ending_here
    end if
    if max_ending_here < 0 then
        max_ending_here ← 0
    end if
end for
return max_so_far
```

Maximum Subarray: Correctness

- Assume the algorithm does not find the maximum subarray sum.
- Let max_so_far be the maximum sum found by Kadane's Algorithm, and let S be the actual maximum subarray sum.
- If $max\_so\_far \neq S$, then there exists a subarray with a sum greater than max_so_far.
- Kadane's Algorithm updates max_so_far whenever max_ending_here exceeds the current max_so_far.
- This means Kadane's Algorithm would have updated max_so_far to S when encountering the subarray with sum S.
- Therefore, max_so_far should have been updated to S, contradicting the assumption.
- Hence, $max\_so\_far = S$, and Kadane's Algorithm correctly finds the maximum subarray sum.

Maximum Subarray: Runtime

- The Divide and Conquer approach has a time complexity of $O(n \log n)$.
- The steps include:
  - Dividing the array: $O(\log n)$ levels of recursion.
  - Finding the maximum sum in each half: $O(n)$ at each level.
- Overall, the algorithm is more efficient than the naive approach for large arrays.
- Kadane's Algorithm, on the other hand, runs in linear time $O(n)$, making it the most efficient solution for this problem.

Question: How does Kadane's Algorithm improve upon the Divide and Conquer approach in terms of complexity and implementation?

Maximum Sub-array: Example Applications

- **Financial Analysis:**
  - Identifying the period with the maximum profit in stock price changes.
  - Example: Finding the period during which a stock's price increased the most.
- **Genomics:**
  - Analyzing DNA sequences to find regions with significant activity or patterns.
  - Example: Identifying the most active region in a sequence of gene expression data.
- **Signal Processing:**
  - Detecting the period with the highest signal strength in time-series data.
  - Example: Finding the time interval with the strongest signal in an audio recording.

Maximum Sub-array: Example Applications

- **Image Processing:**
  - Locating the region with the highest intensity in an image.
  - Example: Detecting the brightest spot in a satellite image.
- **Computer Graphics:**
  - Enhancing regions in a graphical representation.
  - Example: Highlighting areas in a graph with the highest concentration of data points.
- **Gaming:**
  - Calculating the highest score achieved in a game session.
  - Example: Identifying the period during which the player accumulated the highest score.

Integer Multiplication: Background

- The problem we consider is an extremely basic one: the multiplication of two integers.
- In a sense, this problem is so basic that one may not initially think of it even as an algorithmic question.
- But, in fact, elementary schoolers are taught a concrete (and quite efficient) algorithm to multiply two n-digit numbers x and y.
- You first compute a "partial product" by multiplying each digit of y separately by x, and then you add up all the partial products.
- This works for both base-10 and base-2 (i.e., binary) the same.
- The total running time for this algorithm is $O(n^2)$.
    - It takes $O(n)$ time to compute each partial product.
    - There are n partial products.

Integer Multiplication: Example Multiplications

Decimal Multiplication

$$\begin{array}{r} 2384 \\ \times 7433 \\ \hline 7152 \\ 7152\mathit{0} \\ 9536\mathit{00} \\ 16688\mathit{000} \\ \hline\hline \mathbf{17720272} \end{array}$$

Binary Multiplication

$$\begin{array}{r} 1011 \\ \times 1101 \\ \hline 1011 \\ 0000 \\ 1011\mathit{00} \\ 1011\mathit{000} \\ \hline\hline \mathbf{10001111111} \end{array}$$

Integer Multiplication: Karatsuba Algorithm

- One approach to improve the running time of integer multiplication is using a divide and conquer algorithm.
- Karatsuba algorithm is a famous example of this technique:
    1. Split each number into two halves.
    2. Multiply the parts recursively.
    3. Combine the results to get the final product.
- In practice, the performance improvement is only worth it if the number is large enough

Integer Multiplication: Karatsuba Algorithm Pseudo-code

Algorithm 12 Karatsuba Algorithm

Require: Two integers x and y

if $x < 10$ **or** $y < 10$ **then**
    return $x \times y$
end if
$n \leftarrow \max(\text{size of } x, \text{size of } y)$
$m \leftarrow \lceil n/2 \rceil$
$high_1, low_1 \leftarrow \text{split\_at}(x, m)$
$high_2, low_2 \leftarrow \text{split\_at}(y, m)$
$z_0 \leftarrow \text{Karatsuba}(low_1, low_2)$
$z_1 \leftarrow \text{Karatsuba}(low_1 + high_1, low_2 + high_2)$
$z_2 \leftarrow \text{Karatsuba}(high_1, high_2)$
return $(z_2 \times 10^{2 \times m}) + ((z_1 - z_2 - z_0) \times 10^m) + z_0$

Integer Multiplication: Karatsuba Algorithm Example

Step 1: Split the numbers

$$x = 43921 \Rightarrow high_1 = 439, low_1 = 21$$
$$y = 19543 \Rightarrow high_2 = 195, low_2 = 43$$

Step 2: Compute z_0, z_1, z_2

$$z_0 = \text{Karatsuba}(21, 43) \rightarrow 903$$
$$z_1 = \text{Karatsuba}(460, 238) \rightarrow 109480$$
$$z_2 = \text{Karatsuba}(439, 195) \rightarrow 85605$$

Step 3: Combine the results

$$m = \lfloor 5/3 \rfloor$$
$$z_1 - z_2 - z_0 = 109480 - 85605 - 903 = 22972$$
$$\text{Result} = (z_2 \times 10^{2\times 2}) + ((z_1 - z_2 - z_0) \times 10^2) + z_0$$
$$= (85605 \times 10^4) + (22972 \times 10^2) + 903$$
$$= 856050000 + 2297200 + 903$$
$$= 858348103$$

Integer Multiplication: Karatsuba Algorithm Example

Step 2.1: Compute z_0: $\textit{Karatsuba}(21, 43)$

$$
\begin{aligned}
&21 \Rightarrow high_1 = 2, low_1 = 1 \\
&43 \Rightarrow high_2 = 4, low_2 = 3 \\
&z_0 = (2 \times 4 \times 10^{2\times 1}) + ((2+1)(4+3) - 2 \times 4 - 1 \times 3) \times 10^1 + (1 \times 3) \\
&\quad = 903
\end{aligned}
$$

Step 2.2: Compute z_1: $\textit{Karatsuba}(460, 238)$

$$
\begin{aligned}
460 &\Rightarrow high_1 = 46, low_1 = 0 \\
238 &\Rightarrow high_2 = 23, low_2 = 8 \\
z_1 &= 109480
\end{aligned}
$$

Step 2.3: Compute z_2: $\textit{Karatsuba}(439, 195)$

$$
\begin{aligned}
439 &\Rightarrow high_1 = 43, low_1 = 9 \\
195 &\Rightarrow high_2 = 19, low_2 = 5 \\
z_2 &= 85605
\end{aligned}
$$

Integer Multiplication: Runtime

- Naive algorithm runtime: $O(n^2)$.
- The runtime recurrence: $T(n) = 3T(n/2) + O(n)$
- Karatsuba algorithm runtime: $O(n^{\log_2 3}) \approx O(n^{1.585})$.
- Other algorithms:
  - Toom-Cook multiplication: $O(n^{1.465})$.
  - Schönhage-Strassen algorithm: $O(n \log n \log\log n)$.
  - Fastest known algorithm (Fürer's algorithm): $O(n \log n 2^{O(\log^* n)})$.
- Each algorithm provides different trade-offs in terms of implementation complexity and performance.

Integer Multiplication: Takeaway

- Integer multiplication is a fundamental problem with various algorithmic solutions.
- The naive approach is simple but less efficient for large numbers.
- Advanced algorithms like Karatsuba, Toom-Cook, and Schönhage-Strassen offer better performance.
- Understanding these algorithms provides insight into algorithm design and optimization techniques.

Matrix Multiplications: Problem Definition

- Given two $n \times n$ matrices A and B, compute the product matrix $C = A \times B$.
- Traditional matrix multiplication algorithm runs in $O(n^3)$ time.
- Can we use divide and conquer to improve the performance ?

Strassen's Algorithm: Idea

- Strassen's algorithm breaks down one $n \times n$ matrix multiplication into seven $\frac{n}{2} \times \frac{n}{2}$ matrix multiplications.
- This reduces the number of multiplications required compared to the traditional algorithm.
- The algorithm uses the following key identities to achieve this:

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

Strassen's Algorithm: Idea

- The resultant submatrices are then combined to form the final product matrix.

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

$$C_{21} = M_2 + M_4$$

$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Strassen's Algorithm: Pseudo-code

Algorithm 13 Strassen's Algorithm

Require: Two $n \times n$ matrices A and B

Ensure: Product matrix $C = A \times B$

if $n == 1$ **then**
    return $C = A \times B$
end if
Partition A and B into four submatrices of size $\frac{n}{2} \times \frac{n}{2}$
Compute the seven products using Strassen's identities:
$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
$M_2 = (A_{21} + A_{22})B_{11}$
$M_3 = A_{11}(B_{12} - B_{22})$
$M_4 = A_{22}(B_{21} - B_{11})$
$M_5 = (A_{11} + A_{12})B_{22}$
$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$
Compute the resulting submatrices:
$C_{11} = M_1 + M_4 - M_5 + M_7$
$C_{12} = M_3 + M_5$
$C_{21} = M_2 + M_4$
$C_{22} = M_1 - M_2 + M_3 + M_6$
Combine $C_{11}, C_{12}, C_{21}, C_{22}$ into the final matrix C
return C

Strassen's Algorithm: Example

Example: Multiply two 2x2 matrices using Strassen's algorithm

$$A = \begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}, \quad B = \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$

- Compute the seven products:

$$\begin{aligned}
M_1 &= (1+5)(6+2) = 6 \times 8 = 48 \\
M_2 &= (7+5)6 = 12 \times 6 = 72 \\
M_3 &= 1(8-2) = 1 \times 6 = 6 \\
M_4 &= 5(4-6) = 5 \times (-2) = -10 \\
M_5 &= (1+3)2 = 4 \times 2 = 8 \\
M_6 &= (7-1)(6+8) = 6 \times 14 = 84 \\
M_7 &= (3-5)(4+2) = -2 \times 6 = -12
\end{aligned}$$

Strassen's Algorithm: Example

- Compute the resulting submatrices:

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 = 48 - 10 - 8 - 12 = 18 \\
C_{12} &= M_3 + M_5 = 6 + 8 = 14 \\
C_{21} &= M_2 + M_4 = 72 - 10 = 62 \\
C_{22} &= M_1 - M_2 + M_3 + M_6 = 48 - 72 + 6 + 84 = 66
\end{aligned}
$$

- Combine the submatrices into the final matrix:

$$C = \begin{pmatrix} 18 & 14 \\ 62 & 66 \end{pmatrix}$$

Strassen's Algorithm: Runtime

- The traditional matrix multiplication algorithm runs in $O(n^3)$ time.
- Strassen's algorithm divides each $n \times n$ matrix into four $\frac{n}{2} \times \frac{n}{2}$ submatrices.
- Performs seven recursive multiplications on these submatrices.
- Additionally, performs a constant number of matrix additions and subtractions, each taking $O(n^2)$ time.

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

- Strassen's algorithm reduces the time complexity to approximately $O(n^{2.81})$.
- This is achieved by reducing the number of recursive multiplications from 8 to 7.
- The improved runtime comes at the cost of additional additions and subtractions.

Strassen's Algorithm: Takeaway

- Strassen's algorithm demonstrates that matrix multiplication can be performed more efficiently than the traditional $O(n^3)$ approach.
- It laid the groundwork for further research in fast matrix multiplication algorithms.
- Practical implementations need to consider the trade-offs between reduced multiplication operations and increased addition/subtraction operations.
- Strassen's algorithm is especially useful for large matrices where the reduction in multiplication operations significantly improves performance.

Convolution: Problem Definition

- Given two vectors $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$, there are a number of common ways of combining them.
    - One can compute the sum, producing the vector $a + b = (a_0 + b_0, a_1 + b_1, \ldots, a_{n-1} + b_{n-1})$.
    - One can compute the inner product, producing the real number $a \cdot b = a_0 b_0 + a_1 b_1 + \ldots + a_{n-1} b_{n-1}$.
- Another means of combining vectors, very important in applications, is the convolution $a * b$.
- The convolution of two vectors of length n (as a and b are) is a vector with $2n - 1$ coordinates, where coordinate k is equal to:

$$(a * b)_k = \sum_{i=0}^{k} a_i b_{k-i} \quad \text{for} \quad 0 \leq k < 2n - 1$$

Convolutions: Motivating Example 1 - Dice

- Let's say you are throwing a pair of dice and would like to know the chances of getting a certain combination
  - E.g., What are the chances of getting a 1 and 2 ?
- Let's assume the die is fair and it is equally likely to get any of the faces

Die A	1	2	3	4	5	6
Die B	1	2	3	4	5	6

Convolutions: Motivating Example 1 - Dice

2 (1)

Die 1

	1	2	3	4	5	6
1	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
2	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
3	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
4	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
5	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
6	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

Convolutions: Motivating Example 1 - Dice

3 (2)

Die 1

	1	2	3	4	5	6
1	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
2	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
3	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
4	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
5	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
6	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

Convolutions: Motivating Example 1 - Dice

4 (3)

Die 1

	1	2	3	4	5	6
1	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
2	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
3	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
4	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
5	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
6	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

Convolutions: Motivating Example 1 - Dice

5 (4)

Die 1

	1	2	3	4	5	6
1	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
2	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
3	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
4	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
5	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
6	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

Convolutions: Motivating Example 1 - Dice

6 (5)

Die 1

	1	2	3	4	5	6
1	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
2	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
3	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
4	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
5	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
6	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

Convolutions: Motivating Example 1 - Dice

7 (6)

Die 1

	1	2	3	4	5	6
1	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
2	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
3	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
4	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
5	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
6	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

Convolutions: Motivating Example 1 - Dice

8 (5)

Die 1

	1	2	3	4	5	6
1	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
2	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
3	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
4	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
5	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
6	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

Convolutions: Motivating Example 1 - Dice

9 (4)

Die 1

	1	2	3	4	5	6
1	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
2	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
3	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
4	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
5	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
6	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

Convolutions: Motivating Example 1 - Dice

10 (3)

Die 1

	1	2	3	4	5	6
1	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
2	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
3	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
4	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
5	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
6	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

Convolutions: Motivating Example 1 - Dice

11 (2)

Die 1

	1	2	3	4	5	6
1	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
2	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
3	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
4	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
5	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
6	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

Convolutions: Motivating Example 1 - Dice

12 (1)

Die 1

	1	2	3	4	5	6
1	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
2	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
3	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
4	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
5	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
6	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

Convolutions: Motivating Example 1 - Dice

Die 1

	1	2	3	4	5	6
1	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
2	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
3	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
4	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
5	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
6	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

- This takes $O(n^2)$

Convolutions: Motivating Example 1 - Dice

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A	1	2	3	4	5	6
Die B	1	2	3	4	5	6

Convolutions: Motivating Example 1 - Dice

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A							1	2	3	4	5	6
Die B	6	5	4	3	2	1						

Convolutions: Motivating Example 1 - Dice

2 (1)

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A						1	2	3	4	5	6
Die B	6	5	4	3	2	1					

Convolutions: Motivating Example 1 - Dice

3 (2)

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A					1	2	3	4	5	6
Die B	6	5	4	3	2	1				

Convolutions: Motivating Example 1 - Dice

4 (3)

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A				1	2	3	4	5	6
Die B	6	5	4	3	2	1			

Convolutions: Motivating Example 1 - Dice

5 (4)

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A			1	2	3	4	5	6
Die B	6	5	4	3	2	1		

Convolutions: Motivating Example 1 - Dice

6 (5)

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A		1	2	3	4	5	6
Die B	6	5	4	3	2	1	

Convolutions: Motivating Example 1 - Dice

7 (6)

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A	1	2	3	4	5	6
Die B	6	5	4	3	2	1

Convolutions: Motivating Example 1 - Dice

8 (5)

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A	1	2	3	4	5	6	
Die B		6	5	4	3	2	1

Convolutions: Motivating Example 1 - Dice

9 (4)

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A	1	2	3	4	5	6		
Die B			6	5	4	3	2	1

Convolutions: Motivating Example 1 - Dice

10 (3)

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A	1	2	3	4	5	6			
Die B				6	5	4	3	2	1

Convolutions: Motivating Example 1 - Dice

11 (2)

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A	1	2	3	4	5	6				
Die B					6	5	4	3	2	1

Convolutions: Motivating Example 1 - Dice

$\boxed{\mathbf{12}\ (1)}$

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A	1	2	3	4	5	6					
Die B						6	5	4	3	2	1

Convolutions: Motivating Example 1 - Dice

- Another way to look at this operation is as follows:
  - Flip the second list
  - Slide it one step at a time and count the probability

Die A	1	2	3	4	5	6					
Die B						6	5	4	3	2	1

- Both of these examples assume the die is fair
  - i.e, every face of both dice has equal probability of occurrence
- If that is not the case we can just replace the count by multiplication
- This algorithm for computing convolution is $O(n^2)$

Convolutions: Motivating Example 1 - Image Matrix

Image Matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 1 \\ 7 & 8 & 9 & 0 \\ 1 & 3 & 5 & 2 \end{bmatrix}$$

Kernel

$$K = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Convolution

$$A = \begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}$$

Convolutions: Motivating Example 1 - Image Matrix

Image Matrix

$$A = \begin{bmatrix} \color{red}{1} & \color{red}{2} & 3 & 0 \\ \color{red}{4} & \color{red}{5} & 6 & 1 \\ 7 & 8 & 9 & 0 \\ 1 & 3 & 5 & 2 \end{bmatrix}$$

Kernel

$$K = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Convolution

$$A = \begin{bmatrix} 6 & & \\ & & \\ & & \end{bmatrix}$$

Convolutions: Motivating Example 1 - Image Matrix

Image Matrix

$$A = \begin{bmatrix} 1 & \color{red}{2} & \color{red}{3} & 0 \\ 4 & \color{red}{5} & \color{red}{6} & 1 \\ 7 & 8 & 9 & 0 \\ 1 & 3 & 5 & 2 \end{bmatrix}$$

Kernel

$$K = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Convolution

$$A = \begin{bmatrix} 6 & 8 & \\ & & \\ & & \end{bmatrix}$$

Convolutions: Motivating Example 1 - Image Matrix

Image Matrix

$$A = \begin{bmatrix} 1 & 2 & \color{red}{3} & \color{red}{0} \\ 4 & 5 & \color{red}{6} & \color{red}{1} \\ 7 & 8 & 9 & 0 \\ 1 & 3 & 5 & 2 \end{bmatrix}$$

Kernel

$$K = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Convolution

$$A = \begin{bmatrix} 6 & 8 & 6 \\ & & \\ & & \end{bmatrix}$$

Convolutions: Motivating Example 1 - Image Matrix

Image Matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & 0 \\ \color{red}{4} & \color{red}{5} & 6 & 1 \\ \color{red}{7} & \color{red}{8} & 9 & 0 \\ 1 & 3 & 5 & 2 \end{bmatrix}$$

Kernel

$$K = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Convolution

$$A = \begin{bmatrix} 6 & 8 & 6 \\ 12 & & \\ & & \end{bmatrix}$$

Convolutions: Motivating Example 1 - Image Matrix

Image Matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & \color{red}{5} & \color{red}{6} & 1 \\ 7 & \color{red}{8} & \color{red}{9} & 0 \\ 1 & 3 & 5 & 2 \end{bmatrix}$$

Kernel

$$K = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Convolution

$$A = \begin{bmatrix} 6 & 8 & 6 \\ 12 & 14 & \\ & & \end{bmatrix}$$

Convolutions: Motivating Example 1 - Image Matrix

Image Matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & \color{red}{6} & \color{red}{1} \\ 7 & 8 & \color{red}{9} & \color{red}{0} \\ 1 & 3 & 5 & 2 \end{bmatrix}$$

Kernel

$$K = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Convolution

$$A = \begin{bmatrix} 6 & 8 & 6 \\ 12 & 14 & 10 \\ & & \end{bmatrix}$$

Convolutions: Motivating Example 1 - Image Matrix

Image Matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 1 \\ \color{red}{7} & \color{red}{8} & 9 & 0 \\ \color{red}{1} & \color{red}{3} & 5 & 2 \end{bmatrix}$$

Kernel

$$K = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Convolution

$$A = \begin{bmatrix} 6 & 8 & 6 \\ 12 & 14 & 10 \\ 9 & & \end{bmatrix}$$

Convolutions: Motivating Example 1 - Image Matrix

Image Matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 1 \\ 7 & \color{red}{8} & \color{red}{9} & 0 \\ 1 & \color{red}{3} & \color{red}{5} & 2 \end{bmatrix}$$

Kernel

$$K = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Convolution

$$A = \begin{bmatrix} 6 & 8 & 6 \\ 12 & 14 & 10 \\ 9 & 12 & \end{bmatrix}$$

Convolutions: Motivating Example 1 - Image Matrix

Image Matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 1 \\ 7 & 8 & 9 & 0 \\ 1 & 3 & 5 & 2 \end{bmatrix}$$

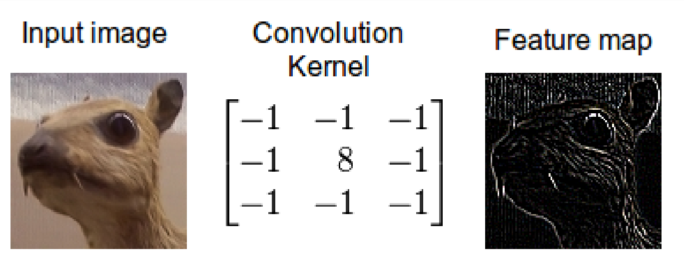
Kernel

$$K = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Convolution

$$A = \begin{bmatrix} 6 & 8 & 6 \\ 12 & 14 & 10 \\ 9 & 12 & 5 \end{bmatrix}$$

Convolutions: Motivating Example 1 - Image Matrix



Convolving an image with an edge detector kernel. [1]

[1] Source: https://developer.nvidia.com/discover/convolution

Convolutions: Problem Definition

- **Convolution:**
  - An operation on two functions f and g producing a third function that expresses how the shape of one is modified by the other.
  - **Discrete Convolution:** Given two sequences a and b, their convolution c is defined as:

$$c[n] = \sum_{m=0}^{n} a[m] \cdot b[n-m]$$

- **Fast Fourier Transform (FFT):**
  - An efficient algorithm to compute the Discrete Fourier Transform (DFT) and its inverse.
  - The DFT of a sequence x of length N is given by:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-i2\pi kn/N}$$

- **Problem:**
  - Compute the convolution of two sequences efficiently using the FFT.
  - Applications in signal processing, image processing, and solving differential equations.

FFT: Pseudocode

Algorithm 14 FFT-based Convolution

Require: Sequences a and b of length N

Ensure: Convolution c of a and b

```
Compute the FFT of a: A ← FFT(a)
Compute the FFT of b: B ← FFT(b)
Multiply pointwise in the frequency domain: C ← A · B
Compute the inverse FFT of C: c ← IFFT(C)
return c
```

FFT: Divide and Conquer Algorithm

- The FFT can be used to compute convolutions efficiently.
- **Steps:**
  - Compute the FFT of both sequences a and b.
  - Multiply the resulting frequency-domain representations element-wise.
  - Compute the inverse FFT of the product to get the convolution result.
- **Time Complexity:** $O(n \log n)$ due to the efficiency of the FFT.

FFT: Example

- Consider two sequences $a = [1, 2, 3]$ and $b = [4, 5, 6]$

FFT: Example

- Consider two sequences $a = [1, 2, 3]$ and $b = [4, 5, 6]$
- Step 1: Compute the FFT of a and b

	Sequence	FFT	Value
a	$[1, 2, 3]$	$A[k]$	{6, -1.5 + 0.87i, -1.5 - 0.87i}
b	$[4, 5, 6]$	$B[k]$	{15, -1.5 + 0.87i, -1.5 - 0.87i}

FFT: Example

- Consider two sequences $a = [1, 2, 3]$ and $b = [4, 5, 6]$
- Step 1: Compute the FFT of a and b

	Sequence	FFT	Value
a	$[1, 2, 3]$	$A[k]$	{6, -1.5 + 0.87i, -1.5 - 0.87i}
b	$[4, 5, 6]$	$B[k]$	{15, -1.5 + 0.87i, -1.5 - 0.87i}

- Step 2: Multiply the FFT results element-wise

	Value
$C[0] = A[0] \cdot B[0]$	90
$C[1] = A[1] \cdot B[1]$	0.75 - 2.6i
$C[2] = A[2] \cdot B[2]$	0.75 + 2.6i

FFT: Example

- Consider two sequences $a = [1, 2, 3]$ and $b = [4, 5, 6]$
- Step 1: Compute the FFT of a and b

	Sequence	FFT	Value
a	$[1, 2, 3]$	$A[k]$	{6, -1.5 + 0.87i, -1.5 - 0.87i}
b	$[4, 5, 6]$	$B[k]$	{15, -1.5 + 0.87i, -1.5 - 0.87i}

- Step 2: Multiply the FFT results element-wise

	Value
$C[0] = A[0] \cdot B[0]$	90
$C[1] = A[1] \cdot B[1]$	0.75 - 2.6i
$C[2] = A[2] \cdot B[2]$	0.75 + 2.6i

- Step 3: Compute the inverse FFT of $C[k]$ to get the convolution result

Convolution Result 18	4	13	28	27

FFT: Correctness

- The correctness of the FFT-based convolution follows from:
  - The correctness of the FFT and its inverse.
  - The convolution theorem, which states that the pointwise product of two sequences' Fourier transforms is the Fourier transform of their convolution.
- Thus, the algorithm correctly computes the convolution by leveraging the FFT.

Question: Why is the convolution theorem crucial in this approach?

FFT: Runtime

- The FFT-based convolution has a time complexity of $O(n \log n)$.
- The steps include:
  - Computing the FFT of both sequences: $O(n \log n)$.
  - Pointwise multiplication: $O(n)$.
  - Computing the inverse FFT: $O(n \log n)$.
- Overall, the algorithm is significantly faster than the brute-force approach for large sequences.

Question: How does the runtime of the FFT-based approach compare to the brute-force solution?

FFT: Application

- **Signal Processing:**
  - Used to filter signals, remove noise, and detect features in time-series data.
- **Image Processing:**
  - Used for image filtering, edge detection, and image convolution.
- **Audio Processing:**
  - Enhances audio signals, equalizes sound frequencies, and applies effects like reverb.
- **Communications:**
  - Used in modulating and demodulating signals in communication systems.
- **Numerical Analysis:**
  - Solves differential equations, convolves functions, and applies integral transforms.
- **Machine Learning:**
  - Applies convolution operations in neural networks, especially in Convolutional Neural Networks (CNNs) for image recognition and classification tasks.

Conclusion

Divide and Conquer Algorithms: Summary

- **Definition:**
  - A strategy to solve a complex problem by breaking it down into simpler sub-problems, solving each sub-problem recursively, and combining their solutions to solve the original problem.
- **Key Steps:**
  - Divide: Break the problem into smaller sub-problems.
  - Conquer: Solve each sub-problem recursively.
  - Combine: Merge the solutions of the sub-problems to form the solution to the original problem.
- **Examples:**
  - Merge Sort: Recursively splits the array in half, sorts each half, and merges the sorted halves.
  - Quick Sort: Partitions the array into sub-arrays around a pivot and recursively sorts the sub-arrays.
  - Binary Search: Recursively divides the search interval in half to find an element in a sorted array.
  - Strassen's Matrix Multiplication: Divides matrices into smaller sub-matrices and combines their products.

Divide and Conquer Algorithms: Summary

- **Applications:**
  - Sorting algorithms (e.g., Merge Sort, Quick Sort)
  - Searching algorithms (e.g., Binary Search)
  - Numerical algorithms (e.g., Fast Fourier Transform)
  - Graph algorithms (e.g., Closest Pair of Points)
- **Advantages:**
  - Often reduces time complexity compared to brute-force approaches.
  - Provides a clear and recursive structure to solve problems.
- **Challenges:**
  - Overhead of recursive calls.
  - Combining solutions of sub-problems can be complex.
- **Conclusion:**
  - Divide and Conquer is a powerful paradigm that provides efficient solutions to many complex problems.
  - Understanding its principles and applications is crucial for designing effective algorithms.

Divide and Conquer Algorithms: Other Examples

- Quicksort
- Power Of Numbers
- K^{th} element of two Arrays
- Cooley–Tukey Fast Fourier Transform (FFT) algorithm
- The Painter's Partition Problem-II
- Modular Exponentiation for large numbers
- Candy
- Sequence of Sequence
- Possible paths
- Scrambled String
- The N^{th} Fibonnaci
- Killing Spree
- Convex Hull

Questions?