CIS 4930, Spring 2022

Divide and Conquer

Viet Tung Hoang

The slides are loosely based on those of Dr Kevin Wayne, Princeton University and Prof. Mary Wootters, Stanford University.
Divide & Conquer Paradigm

- Divide the problem into several subproblems
- Solve (conquer) each subproblem recursively
- Combine solution of subproblems into overall solutions
Divide & Conquer Paradigm

Most common usage:
- Divide to **two** subproblems of size $n / 2$
- Combine using $O(n)$ time

**Brute-force:** $\Theta(n^2)$  
**Divide-and-conquer:** $\Theta(n \log(n))$
Agenda

1. Merge Sort

2. Counting Inversions

3. Maximum Subarray
Sorting

Given a list of $n$ elements, rearrange them in ascending order
Sorting Applications

Some problems are easier once elements are sorted:

- Binary search in a database
- Remove duplicate elements in a list

Non-obvious applications:

- Closest Pair of Points
- Convex Hull
Merge Sort

Recursive magic!

Recursive magic!
Merge Sort

Recursive magic!

Merge
Merge Sort

Recursive magic!

Merge
Merge Sort

Recursive magic!

Merge
Merge Sort

Recursive magic!

Merge

Recursive magic!
Merge Sort

Recursive magic!

Merge
Merge Sort

6 4 3 8 1 5 2 7

Recursive magic!

3 4 6 8

Recursive magic!

1 2 5 7

Merge

1 2 3
Merge Sort

Recursive magic!

Merge
Merge Sort

Recursive magic!

Merge
Merge Sort

Recursive magic!

Merge
Merge Sort

Recursive magic!

Merge
Merge Sort

Recursive magic!

Merge
Merge Sort

Recursive magic!

Recursive magic!

Merge
Merge Sort

Recursive magic!

Merge

Recursive magic!
Merge Sort

Recursive magic!

Recursive magic!

Merge
Merge Sort

Recursive magic!

Merge
Recursive magic!

Recursive magic!

Merge
Merge Sort

Recursion magic!

Merge
Merge Sort Implementation

```plaintext
procedure MergeSort(A[1 : n])
if n = 1 then return A  // Base case
MergeSort(A[1 : n/2])  // Sort left half
MergeSort(A[n/2 + 1 : n])  // Sort right half
B ← Merge(A[1 : n/2], A[n/2 + 1 : n])
return B
```

How Long Does It Take To Merge?

Need $\Theta(k)$ time to merge two sorted lists of size $k/2$:
- After each comparison, always move a pointer one step ahead.

$k/2$

3 4 6 8

Merge

1 2 3 4 5 6 7 8

$k/2$

1 2 5 7
Running-time Analysis

$T(n)$: maximum number of steps to Merge-Sort an array of size $n$

Merge-Sort recurrence:

$$T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases}$$
Recursion Tree

\[ T(n) = 2T(n/2) + n \text{ and } T(1) = 1 \]

\[ 2^{\log_2(n)} = n \]

\[ T(n) = \Theta(n \log(n)) \]
Agenda

1. Merge Sort

2. Counting Inversions

3. Maximum Subarray
Counting Inversions

**Want**: match your song preferences with others to recommend new songs

Your ranking of $n$ songs $\rightarrow$ Database $\rightarrow$ People with similar taste

Start listening for free.

Enter an artist, song, or genre to create a station.

Every Pandora station evolves with your tastes. Sit back and enjoy.
Measuring Similarity

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>me</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>you</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

**Inversion**: I rank B higher than D, but you disagree

**Similarity metric**: the number of inversions between two rankings
Counting Inversions

Let \( a_1, \ldots, a_n \) be a permutation of \( 1, \ldots, n \).

\((a_i, a_j)\) is an inversion if \( i < j \) but \( a_i > a_j \).

There are 10 inversions here:

\( (4, 2), (4, 3), (5, 2), (5, 3), (5, 4), (6, 3), (8, 2), (8, 3), (8, 6), (8, 7) \)

**Want:** count the total number of inversions

**Brute-force:** check all \( O(n^2) \) pairs
Counting Inversions: Divide & Conquer

Count inversions in left half

1 5 4 8
(5, 4)

Count inversions in right half

2 6 3 7
(6, 3)

Count inversions across two halves:

(4, 2), (4, 3), (5, 2), (5, 3), (8, 2), (8, 3), (8, 6), (8, 7)

Totally $1 + 1 + 8 = 10$ inversions
Counting Inversions Across Two Halves

Sorting the subarrays doesn’t change the number of crossing inversions

For example, \((4, 3)\) remains an inversion after sorting
Warm-up Algorithm

1 5 4 8

2 6 3 7

Sort two subarrays

1 4 5 8

2 3 6 7
Warm-up Algorithm

Binary search

3 elements

2

3 inversions of form \((a, 2)\)
Warm-up Algorithm

1 4 5 8

binary search

3 elements

2 3 inversions of form \((a, 2)\)

3 3 inversions of form \((a, 3)\)
Warm-up Algorithm

1 element

2 \rightarrow \text{3 inversions of form } (a, 2)

3 \rightarrow \text{3 inversions of form } (a, 3)

6 \rightarrow \text{1 inversion of form } (a, 6)

binary search
Warm-up Algorithm

2 → 3 inversions of form \((a, 2)\)
3 → 3 inversions of form \((a, 3)\)
6 → 1 inversion of form \((a, 6)\)
7 → 1 inversion of form \((a, 7)\)

Binary search

1 element

1
4
5
8

2 3 6 7

Totally \(3 + 3 + 1 + 1 = 8\) inversions
Issues in Warm-up Algorithm

**Combing cost:**
- Sorting requires $\Theta(n \log(n))$ time
- Doing $n$ binary searches need $\Theta(n \log(n))$ time

**Want:** $O(n)$ time for combining cost
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: \( O(n) \) time

Merge-and-Count
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

Merge-and-Count
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

3 elements

1 4 5 8

2 3 6 7

Merge-and-Count

1
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

Merge-and-Count
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

Merge-and-Count

1 2
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

Merge-and-Count
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

Merge-and-Count
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

Merge-and-Count
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

Merge-and-Count

1  2  3  4  5

1  4  5  8

2  3  6  7
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

Merge-and-Count

1 2 3 4 5
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

Merge-and-Count

1 4 5 8

1 element

2 3 6 7

3 3

1 2 3 4 5
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

1 element

Merge-and-Count

1 2 3 4 5 6
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

Merge-and-Count
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

Merge-and-Count

1 2 3 4 5 6
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

1 element

Merge-and-Count
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

1 4 5 8

2 3 6 7

Merge-and-Count

1 2 3 4 5 6 7
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

Merge-and-Count
How to Avoid Binary Searches

Assuming that subarrays are sorted

Count via Merge algorithm in Merge Sort: $O(n)$ time

Merge-and-Count

Totally $3 + 3 + 1 + 1 = 8$ inversions
How to Eliminate Sorting in Combining

**Inventor’s Paradox:** “The more ambitious plan may have more chances of success” — George Polya

**Input:**

```
1 5 4 8 2 6 3 7
```

**Sort-and-Count**

**Output:**

```
1 2 3 4 5 6 7 8
```

10 inversions
Divide and Conquer for Sort-and-Count

1  5  4  8  2  6  3  7

1  5  4  8
2  6  3  7

Sort-and-Count in left half

1  4  5  8
2  3  6  7

1 inversion
1 inversion

Merge-and-Count

1  2  3  4  5  6  7  8

8 inversions

Totally 1 + 1 + 8 = 10 inversions
Counting Inversions Implementation

procedure Sort-and-Count(A[1 : n])
if n = 1 then return (0, A)  // Base case
(v₀, B₀) ← Sort-and-Count(A[1 : n/2])
(v₁, B₁) ← Sort-and-Count(A[n/2 + 1 : n])
(v₂, B₂) ← Merge-and-Count(B₀, B₁)
return (v₀ + v₁ + v₂, B₂)

T(n): maximum number of steps to Sort-and-Count an array of size n

Sort-and-Count recurrence:

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{otherwise} 
\end{cases} \]

Recursion tree:

\[ T(n) = \Theta(n \log(n)) \]
Agenda

1. Merge Sort

2. Counting Inversions

3. Maximum Subarray
A Motivating Application

How to find the brightest region in the picture?
From Image to Array

Relative brightness, compared to an average spot

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>2</td>
<td>-3</td>
<td>5</td>
<td>-4</td>
<td>-8</td>
<td>3</td>
<td>-3</td>
</tr>
<tr>
<td>2</td>
<td>-4</td>
<td>-6</td>
<td>-8</td>
<td>2</td>
<td>-5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-2</td>
<td>9</td>
<td>-9</td>
<td>-1</td>
<td>10</td>
<td>-5</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>-3</td>
<td>5</td>
<td>-7</td>
<td>8</td>
<td>-2</td>
<td>2</td>
<td>-6</td>
</tr>
</tbody>
</table>

Goal: Find a subarray of maximum sum

Brightest area
Maximum Subarray Problem

Given an array of numbers, find a subarray of maximum sum.

The general problem considers 2D array, but here we only study the 1D one.
Brute-force Solution

**Brute-force**: check all $\Theta(n^2)$ subarrays

```
for start ← 1 to n do
    for finish ← start to n do
        $S[start, finish] \leftarrow A[start] + \cdots + A[finish]$
```

**Goal**: Implement the inner loop in just $O(n)$ time

Improved brute-force takes $O(n^2)$ time
Computing Sums Quickly

start = 3

finish = 3

\[
\begin{array}{cccccc}
-2 & 1 & 6 & -2 & 3 & 1 \\
\end{array}
\]

\[S_0 = A[3] = 6\]

finish = 4

\[
\begin{array}{cccccc}
-2 & 1 & 6 & -2 & 3 & 1 \\
\end{array}
\]

\[S_1 = S_0 + A[4] = 4\]

finish = 5

\[
\begin{array}{cccccc}
-2 & 1 & 6 & -2 & 3 & 1 \\
\end{array}
\]

\[S_2 = S_1 + A[5] = 7\]

finish = 6

\[
\begin{array}{cccccc}
-2 & 1 & 6 & -2 & 3 & 1 \\
\end{array}
\]

\[S_3 = S_2 + A[6] = 8\]

Running time is \(\Theta(n)\)
Maximum Subarray: Divide & Conquer

Find maximum subarray in left half

Find maximum subarray in right half

Want: Find the maximum subarray crossing the two halves

start \leq n/2

finish \geq n/2 + 1

Overall optimal solution would be the best among the three
Maximum Crossing Subarray

Find start to maximize $A[\text{start}] + \cdots + A[\lfloor n/2 \rfloor]$

Find finish to maximize $A[\lceil n/2 + 1 \rceil] + \cdots + A[\text{finish}]$

Brute-force: $\Theta(n)$ time
Maximum Subarray: Implementation

procedure Maximum-Subarray(A[1 : n])
if \( n = 1 \) then return (1, 1, A[1])  // Base case

(start\(_0\), finish\(_0\), v\(_0\)) ← Maximum-Subarray(A[1 : n/2])

(start\(_1\), finish\(_1\), v\(_1\)) ← Maximum-Subarray(A[n/2 + 1 : n])

start\(_1\) ← start\(_1\) + n/2; finish\(_1\) ← finish\(_1\) + n/2

(start\(_2\), finish\(_2\), v\(_2\)) ← Maximum-Crossing-Subarray(A, min, max)

Pick \( b \) so that \( v_b = \max\{v_0, v_1, v_2\} \)

return (start\(_b\), finish\(_b\), v\(_b\))

\( T(n) \): max number of steps to do Maximum-Subarray on an array of size \( n \)

Maximum-Subarray recurrence:

\[
T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{otherwise}
\end{cases}
\]