CSCI3160 Design and Analysis of Algorithms (2025 Fall)

Divide and Conquer

Instructor: Xiao Liang[1]

Department of Computer Science and Engineering
Chinese University of Hong Kong

[1]These slides are primarily based on materials prepared by Prof. Yufei Tao (please refer to Prof. Tao's version from 2024 Fall for the original content). Some modifications have been made to better align with this year's teaching progress, incorporating student feedback, in-class interactions, and my own teaching style and research perspective.

In this lecture, we will discuss the **divide and conquer** technique for designing algorithms with strong performance guarantees. Our discussion will be based on the following problems:

1. Sorting (a review of merge sort)
2. Counting inversions
3. Dominance counting
4. Matrix multiplication

Recall: Principle of recursion

> When dealing with a subproblem (same problem but with a smaller input), consider it solved, and use the subproblem's output to continue the algorithm design.

- When dividing, we utilize recursion to reduce the original problem into subproblems.
- When conquering, we tackle the **core problem** "hidden within" the original problem.

Sorting

Sorting

Problem: Given an array A of n distinct integers, produce another array where the same integers have been arranged in ascending order.

The Merge Sort Algorithm:

- **Divide:** Let A_1 the array containing the first $\lceil n/2 \rceil$ elements of A, and A_2 be the array containing the other elements of A.
  Sort A_1 and A_2 recursively.
- **Conquer:** Merge the two sorted arrays A_1 and A_2 in ascending order. This can be done in $O(n)$ time.

Sorting

Running Time: Let $f(n)$ denote the worst-case cost of the algorithm on an array of size n. Then:

$$f(n) \quad \leq \quad 2 \cdot f(\lceil n/2 \rceil) + O(n)$$

which gives $f(n) = O(n \log n)$.

Counting Inversions

Counting Inversions

Let: A = an array of n distinct integers.

An **inversion** is a pair of (i, j) such that

- $1 \leq i < j \leq n$, and
- $A[i] > A[j]$.

Example: Consider $A = (10, 3, 9, 8, 2, 5, 4, 1, 7, 6)$.
Then $(1, 2)$ is an inversion because $A[1] = 10 > A[2] = 3$. So are $(1, 3), (3, 4), (4, 5)$, and so on.
There are in total 29 inversions.

Think: Can you come up with a naive algorithm that solves this problem in time $O(n^2)$?

Answer: Compare for each pair one by one in order. It takes

$$(n-1) + (n-2) + \ldots + 1 = O(n^2)$$

steps.

Counting Inversions

Problem: Given an array A of n distinct integers, count the number of inversions.

We will do in the class: $O(n \log^2 n)$ time.
You will do as an exercise: $O(n \log n)$ time.

Counting Inversions

- **Divide:** Let A_1 the array containing the first $\lceil n/2 \rceil$ elements of A, and A_2 be the array containing the other elements of A.

  Solve the "counting inversions" problem recursively on A_1 and A_2, respectively. By doing so, we have already obtained the number m_1 of inversions in A_1, and similarly, the number m_2 for A_2.

- **Conquer:**
  It remains to count the number of **crossing inversions** (i, j) where $i \in A_1$ and $j \in A_2$.

Counting Inversions

A_1 = the array containing the first $\lceil n/2 \rceil$ elements of A
A_2 = the array containing the other elements of A.

Next, perform the following two steps:

1. Sort A_1.
   - in $O(n \log n)$ using merge sort.
2. For each element $e \in A_2$, count how many crossing inversions e produces using **binary search**.
   - Since $|A_2| = \frac{n}{2}$, there are $n/2$ binary searches performed in total, which takes $O(n \log n)$ time.

Illustration by Example

Example (cont.): $A = (10, 3, 9, 8, 2, 5, 4, 1, 7, 6)$.
$A_1 = (2, 3, 8, 9, 10)$ (sorted), $A_2 = (5, 4, 1, 7, 6)$

Element 5 produces 3 crossing inversion
Element 4 produces 3, too.
Elements 1, 7, and 6 produce 5, 3, and 3 crossing inversions, respectively.

- **Think:** How to obtain each count with binary search?

Counting Inversions

Running Time: Let $f(n)$ denote the worst-case cost of the algorithm on an array of size n. Then:

$$f(n) \quad \leq \quad 2 \cdot f(\lceil n/2 \rceil) + O(n \log n)$$

which gives $f(n) = O(n \log^2 n)$.

Dominance Counting

Dominance Counting

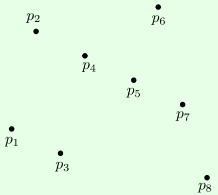
Denote by $\mathbb{Z}$ the set of integers.
Given a point p in two-dimensional space $\mathbb{Z}^2$, denote by $p[1]$ and $p[2]$ its x- and y-coordinate, respectively.

Given two distinct points p and q, we say that q **dominates** p if $p[1] \le q[1]$ and $p[2] \le q[2]$; see the figure below:

Dominance Counting

Let P be a set of n points in $\mathbb{Z}^2$ with distinct x-coordinates. Find, for each point $p \in P$, the number of points in P that are dominated by p.

Example:



We should output: $(p_1, 0), (p_2, 1), (p_3, 0), (p_4, 2), (p_5, 2), (p_6, 5), (p_7, 2), (p_8, 0)$.

Dominance Counting

Let P be a set of n points in $\mathbb{Z}^2$ with distinct x-coordinates. Find, for each point $p \in P$, the number of points in P that are dominated by p.

We will do in the class: $O(n \log^2 n)$ time.
You will do as an exercise: $O(n \log n)$ time.

Note: We can assume without loss of generality that the points are given in ascending order in their x-coordinates, i.e.,

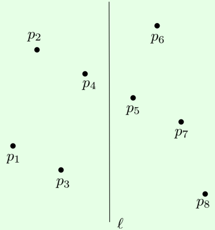
$$p_1[1] < p_2[1] < \ldots < p_n[1].$$

This is without loss of generality because we can always sort them to be so in $O(n \log n)$ time, and our dominance counting algorithm anyway won't be faster than that ($O(n \log^2 n)$ or $O(n \log n)$).

Dominance Counting

Divide: Find a vertical line ℓ such that P has $\lceil n/2 \rceil$ points on each side of the line.

Example:

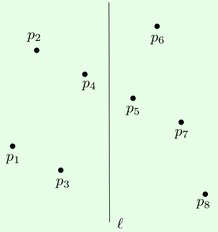


Think: How to find such ℓ in $O(n \log n)$ time? How about $O(n)$ time?

Dominance Counting

Divide:
P_1 = the set of points of P on the left of ℓ
P_2 = the set of points of P on the right of ℓ

Example:



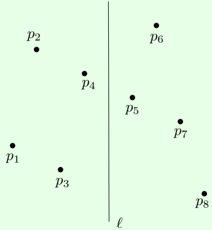
$$P_1 = \{p_1, p_2, p_3, p_4\}$$
$$P_2 = \{p_5, p_6, p_7, p_8\}.$$

Dominance Counting

Divide:

Solve the dominance counting problem on P_1 and P_2 separately.

Example:



On P_1, we have obtained: $(p_1, 0), (p_2, 1), (p_3, 0), (p_4, 2)$.

On P_2, we have obtained: $(p_5, 0), (p_6, 1), (p_7, 0), (p_8, 0)$.

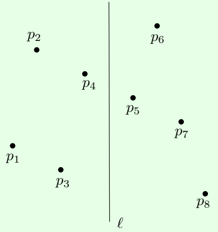
The counts obtained for the points in P_1 are final (**think:** why?).

Dominance Counting

Conquer:

It remains to count, for each point $p_2 \in P_2$, how many points in P_1 it dominates.

Example:



On P_2, we have obtained: $(p_5, 0), (p_6, 1), (p_7, 0), (p_8, 0)$.

Regarding p_5, for example, we still need to find out that it dominates 2 points from P_1.

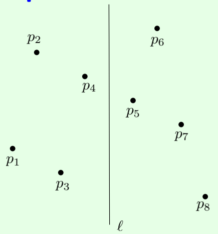
The x-coordinates do not matter any more!

Dominance Counting

Conquer:

Sort P_1 by **y-coordinate**.
Then, for each point $p_2 \in P_2$, we can obtain the number points in P_1 dominated by p_2 using binary search.

Example:



P_1 in ascending of y-coordinate: p_3, p_1, p_4, p_2.

How to perform binary search to obtain the fact that p_5 dominates 2 points in P_1?

- Search using the y-coordinate of p_5.

Dominance Counting

Analysis:

Let $f(n)$ be the worst-case running time of the algorithm on n points. Then:

$$f(n) \le 2f(\lceil n/2 \rceil) + O(n \log n)$$

which solves to $f(n) = O(n \log^2 n)$.

Matrix Multiplication

Matrix Multiplication

Problem: Given two $n \times n$ matrices A and B, compute their product AB.

We store an $n \times n$ matrix with an array of length n^2 in "row-major" order.

Example: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ is stored as $(1, 2, 3, 4)$.

Note that any $A[i, j]$ — the element of A at the i-th row and j-th column — can be accessed in $O(1)$ time.

Trivial: $O(n^3)$ time
We will do in the class: $O(n^{2.81})$ time for n being a power of 2
You will do as an exercise: $O(n^{2.81})$ time for any n.

Matrix Multiplication

Warm Up: Suppose we want to compute $\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix}$. How many multiplication operations do we need to perform?

Trivial: 8.

Non-trivial: 7.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{bmatrix}$$

where

$$\begin{aligned} p_1 &= a(f - h) \\ p_2 &= (a + b)h \\ p_3 &= (c + d)e \\ p_4 &= d(g - e) \\ p_5 &= (a + d)(e + h) \\ p_6 &= (b - d)(g + h) \\ p_7 &= (a - c)(e + f) \end{aligned}$$

Matrix Multiplication (Strassen's Algorithm)

Recall that the input A and B are order-n (i.e., $n \times n$) matrices. Assume for simplicity that n is a power of 2. Divide each of A and B into 4 submatrices of order $n/2$:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

It is easy to verify:

$$AB = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

How many order-$(n/2)$ matrix multiplications do we need?

Trivial: 8.
Non-trivial: 7 — see the next slide.

Matrix Multiplication

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{bmatrix}$$

$$\begin{aligned}
p_1 &= A_{11}(B_{12} - B_{22}) \\
p_2 &= (A_{11} + A_{12})B_{22} \\
p_3 &= (A_{21} + A_{22})B_{11} \\
p_4 &= A_{22}(B_{21} - B_{11}) \\
p_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
p_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
p_7 &= (A_{11} - A_{21})(B_{11} + B_{12})
\end{aligned}$$

If $f(n)$ is the worst-case time of computing the product of two order-n matrices, then each of p_i $(1 \leq i \leq 7)$ can be computed in $f(n/2) + O(n^2)$ time, where

- $f(n/2)$: each p_i contains exactly one multiplication between two order-$\frac{n}{2}$ matrices.
- $O(n^2)$: each p_i contains a constant number of addition/deduction between two order-$\frac{n}{2}$ matrices.

Matrix Multiplication

Therefore:

$$f(n) \leq 7f(n/2) + O(n^2)$$

which solves to $f(n) = O(n^{\log_2 7}) = O(n^{2.81})$.

Note: This can be derived using the Master Theorem. If you need a refresher, please review it. See Sections 4.2–4.5 of [CLRS].

Who Does the Heavy Lifting in Divide and Conquer?

Some students asked about certain seemingly odd aspects of divide-and-conquer (and recursive) algorithms. The question can be summarized as follows:

- The intermediate steps in these algorithms often feel "hollow," making it unclear which part of the implementation is doing the real work.

This is a perfectly reasonable concern and a common source of confusion for beginners.

The short answer:

- Yes, in divide-and-conquer algorithms, the heavy lifting is done by:
  - the **conquer** step (combining solutions), and
  - the **base cases** (which terminate recursion).

And yes, the intermediate steps in these algorithms can be interpreted as being "hollow."

Who Does the Heavy Lifting in Divide and Conquer?

Example: Fibonacci

Naive (incorrect) recursion

```
int fibonacci(int n) {
  return fibonacci(n-1) + fibonacci(n-2); // no base cases!
}
```

Correct recursive definition

```
int fibonacci(int n) {
  if (n == 0) return 0;         // base case
  if (n == 1) return 1;         // base case
  return fibonacci(n-1) + fibonacci(n-2); // conquer (combine)
}
```

- The *conquer* step is the combination `fibonacci(n-1) + fibonacci(n-2)`.
- The *base cases* ensure termination; once reached, the recursion stack unwinds and the combinations propagate results upward.

Who Does the Heavy Lifting? — Merge Sort

- Heavy lifting:
  - **Base case**: size ≤ 1 (already sorted).
  - **Conquer step**: *merge* two sorted halves.

Recursive structure

MergeSort(A)

if $|A| \leq 1$: return A (base case)
Split A into L and R (roughly equal sizes)
L' $\leftarrow$ MergeSort(L)
R' $\leftarrow$ MergeSort(R)
return Merge(L', R') (conquer)

The Merge$(\cdot, \cdot)$ function is shown on the next slide.

Who Does the Heavy Lifting? — Merge Sort

Merge(L, R)

Initialize empty list S
While L and R are nonempty: append smaller front to S
Append any remaining items of L, then of R, to S
Return S