

## STRASSEN'S MATRIX MULTIPLICATION

Strassen's Matrix Multiplication is the divide and conquer approach to solve the matrix multiplication problems. The usual matrix multiplication method multiplies each row with each column to achieve the product matrix. The time complexity taken by this approach is  $O(n^3)$ , since it takes two loops to multiply. Strassen's method was introduced to reduce the time complexity from  $O(n^3)$  to  $O(n^{\log 7})$ .

### Strassen's Matrix Multiplication Formula:

Given two matrices A and B, and dividing them into submatrices:

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

Strassen's algorithm computes 7 products as follows:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_2 = (A_{21} + A_{22})B_{11}$$

$$M_3 = A_{11}(B_{12} - B_{22})$$

$$M_4 = A_{22}(B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{12})B_{22}$$

$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

The final matrix  $C=AB$  is computed as:

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

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

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

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

**Example: 2x2 Matrix Multiplication using Strassen's Algorithm**

### Strassen's Algorithm Implementation With Examples

Strassen's algorithm speeds up matrix multiplication, particularly for large matrices. Let's explore how the algorithm divides the matrix into submatrices and recursively calculated.

#### Mechanism:

1. **Dividing Matrices into Submatrices:** Divide the matrices A and B into 4 submatrices each (quadrants).
2. **Recursive Decomposition:** Recursively apply the same procedure for smaller submatrices until the base case is reached (2x2 matrices).
3. **Constructing the Resultant Matrix:** Combine the computed products to obtain the final result.

4. **Base Case Handling:** For small matrices (2x2), use traditional matrix multiplication directly.
5. **Optimizations:** Strassen's method reduces computational overhead by cutting down multiplication steps.

### Example problem:

Given Matrices:

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

We divide the matrices into submatrices:

$$A_{11} = 1, A_{12} = 2, A_{21} = 3, A_{22} = 4$$

$$B_{11} = 5, B_{12} = 6, B_{21} = 7, B_{22} = 8$$

Compute the 7 products:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) = (1 + 4)(5 + 8) = 5 \times 13 = 65$$

$$M_2 = (A_{21} + A_{22})B_{11} = (3 + 4) \times 5 = 7 \times 5 = 35$$

$$M_3 = A_{11}(B_{12} - B_{22}) = 1 \times (6 - 8) = 1 \times (-2) = -2$$

$$M_4 = A_{22}(B_{21} - B_{11}) = 4 \times (7 - 5) = 4 \times 2 = 8$$

$$M_5 = (A_{11} + A_{12})B_{22} = (1 + 2) \times 8 = 3 \times 8 = 24$$

$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12}) = (3 - 1) \times (5 + 6) = 2 \times 11 = 22$$

$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22}) = (2 - 4) \times (7 + 8) = (-2) \times 15 = -30$$

Now, compute the four submatrices of the resulting matrix CC:

$$C_{11} = M_1 + M_4 - M_5 + M_7 = 65 + 8 - 24 - 30 = 19$$

$$C_{21} = M_2 + M_4 = -2 + 24 = 22$$

$$C_{21} = M_2 + M_4 = 35 + 8 = 43$$

$$C_{22} = M_1 + M_3 - M_2 + M_6 = 65 - 2 - 35 + 22 = 50$$

The resulting matrix C is:

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

This method reduces the number of multiplications, optimizing performance.

### Advantages and Limitations of Strassen's Algorithm

Although Strassen's algorithm is effective for large matrices, it becomes less efficient for small matrices (e.g.,  $n < 32$ ) due to higher overhead. The algorithm's recursive decomposition can cause numerical instability from floating-point precision errors.

Additionally, Strassen increases memory usage, making it less suitable for memory-constrained environments or when high precision is needed. For small matrices, traditional methods are often more efficient due to lower overhead.

BENEFITS	DRAWBACKS
Enhanced Computational Efficiency: Reduces multiplication operations, making it faster for large matrices.	High Memory Consumption: Requires extra memory for recursive submatrix storage.
Reduced Computational Complexity: Improves time complexity to $O(n^{2.81})$ , faster than traditional methods.	Challenges in Real-World Implementation: Complex to implement, especially for non-square or very large matrices.
Optimized Execution Speed: Processes large matrices significantly faster, ideal for high-performance computing.	Issues with Numerical Precision: Prone to instability, particularly with matrices containing extreme values.
Mathematical Elegance and Innovation: Introduces a recursive method to reduce the number of multiplications.	Limited Practical Use Cases: Not suitable for smaller matrices or scenarios where memory is limited.

This method is used recursively to perform the seven  $(n/2) \times (n/2)$  matrix multiplications, then the recurrence equation for the number of scalar multiplications performed is:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 7 T(n/2) \end{aligned}$$

Solving this for the case of  $n = 2^k$  is easy:

$$\begin{aligned} T(2^k) &= 7 T(2^{k-1}) \\ &= 7^2 T(2^{k-2}) \\ &= \dots \\ &= 7^i T(2^{k-i}) \end{aligned}$$

$$\begin{aligned} \text{Put } i &= k \\ &= 7^k T(1) \\ &= 7^k \end{aligned}$$

$$\begin{aligned} \text{That is, } T(n) &= 7^{\log_2 n} \\ &= n^{\log_2 7} \\ &= O(n^{\log_2 7}) = O(n^{2.81}) \end{aligned}$$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves the hundreds.

**FINDING MIN MAX**

Max-Min problem is to find a maximum and minimum element from the given array. We can effectively solve it using divide and conquer approach. In the traditional approach, the maximum and minimum element can be found by comparing each element and updating Max and Min values as and when required. This approach is simple, but it does  $(n - 1)$  comparisons for finding max and the same number of comparisons for finding the min. It results in a total of  $2(n - 1)$  comparisons. Using a divide and conquer approach, we can reduce the number of comparisons. Divide and conquer approach for Max. Min problem works in three stages.

- If  $a_1$  is the only element in the array,  $a_1$  is the maximum and minimum.
- If the array contains only two elements  $a_1$  and  $a_2$ , then the single comparison between two elements can decide the minimum and maximum of them.
- If there are more than two elements, the algorithm divides the array from the middle and creates two subproblems. Both subproblems are treated as an independent problem and the same recursive process is applied to them. This division continues until subproblem size becomes one or two.

After solving two subproblems, their minimum and maximum numbers are compared to build the solution of the large problem. This process continues in a bottom-up fashion to build the solution of a parent problem.

```
BEGIN
READ n
READ array elements
CALL FIND_MIN_MAX(array, 0, n - 1)
PRINT "Minimum element =", minimum
PRINT "Maximum element =", maximum
END
```

**PROCEDURE FIND\_MIN\_MAX(array, low, high)**

```
IF low = high THEN
    minimum ← array[low]
    maximum ← array[low]
    RETURN (minimum, maximum)
ELSE IF high = low + 1 THEN
    IF array[low] < array[high] THEN
```

```

        minimum ← array[low]
        maximum ← array[high]
ELSE
        minimum ← array[high]
        maximum ← array[low]
END IF
RETURN (minimum, maximum)
ELSE
        mid ← (low + high) DIV 2
        (min1, max1) ← FIND_MIN_MAX(array, low, mid)
        (min2, max2) ← FIND_MIN_MAX(array, mid + 1, high)
IF min1 < min2 THEN
        minimum ← min1
ELSE
        minimum ← min2
END IF
IF max1 > max2 THEN
        maximum ← max1
ELSE
        maximum ← max2
END IF
RETURN (minimum, maximum)
END IF
END PROCEDURE

```

The conventional algorithm takes  $2(n - 1)$  comparisons in worst, best and average case. DC\_MAXMIN does two comparisons to determine the minimum and maximum element and creates two problems of size  $n/2$ , so the recurrence can be formulated as

$$T(n) = 0, \text{ if } n = 1$$

$$T(n) = 1, \text{ if } n = 2$$

$$T(n) = 2T(n/2) + 2, \text{ if } n > 2$$

Let us solve this equation using interactive approach.

$$T(n) = 2T(n/2) + 2 \dots \textbf{(1)}$$

By substituting  $n$  by  $(n / 2)$  in Equation (1)

$$T(n/2) = 2T(n/4) + 2$$

$$\Rightarrow T(n) = 2(2T(n/4) + 2) + 2$$

$$= 4T(n/4) + 4 + 2 \dots \mathbf{(2)}$$

By substituting  $n$  by  $n/4$  in Equation (1),

$$T(n/4) = 2T(n/8) + 2$$

Substitute it in Equation (1),

$$T(n) = 4[2T(n/8) + 2] + 4 + 2$$

$$= 8T(n/8) + 8 + 4 + 2$$

$$= 2^3 T(n/2^3) + 2^3 + 2^2 + 2^1$$

...

...

After  $k - 1$  iterations

$$\begin{aligned} \therefore T(n) &= 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + 2^{k-1} + 2^{k-2} + \dots \\ &\quad + 2^3 + 2^2 + 2^1 \\ &= 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + \sum_{i=1}^{k-1} 2^i \\ &= 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + (2^k - 2) \end{aligned}$$

$$\text{Let } n = 2^k \Rightarrow 2^{k-1} = \left(\frac{n}{2}\right)$$

$$\begin{aligned} \therefore T(n) &= \left(\frac{n}{2}\right) T\left(\frac{2^k}{2^{k-1}}\right) + (n - 2) \\ &= \left(\frac{n}{2}\right) T(2) + (n - 2) \end{aligned}$$

For  $n = 2$ ,  $T(n) = 1$  (two elements require only 1 comparison to determine min-max)

$$T(n) = \left(\frac{n}{2}\right) + (n - 2) = \left(\frac{3n}{2} - 2\right)$$

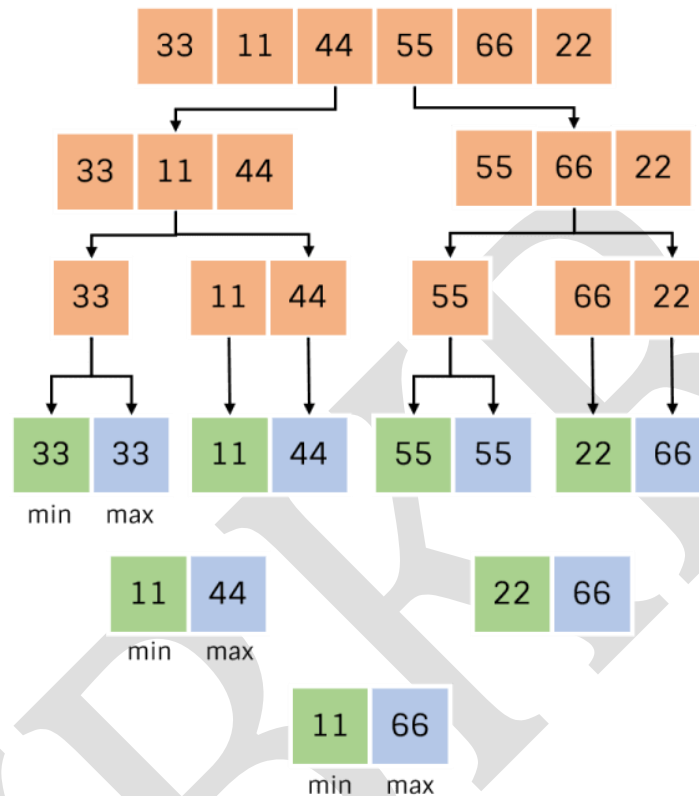
It can be observed that divide and conquer approach does only  $[(3n/2) - 2]$  comparisons compared to  $2(n - 1)$  comparisons of the conventional approach.

For any random pattern, this algorithm takes the same number of comparisons.

It can be observed that divide and conquer approach does only comparisons compared to  $2(n - 1)$  comparisons of the conventional approach. For any random pattern, this algorithm takes the same number of comparisons.

### Problem: Find max and min from the sequence {33, 11, 44, 55, 66, 22} using divide and conquer approach

During the divide step, the algorithm divides the array until it reaches a size of one or two. Once the array size reaches the base case, we may get the maximum and minimum number from each array recursively. This method is continued until all of the subarrays have been processed. The figure below depicts the complete procedure.



The orange cell represents the stage of stepwise division. In the conquer stage, the smallest element from the parent arrays is placed in the green cell, while the largest element is stored in the blue cell.

#### Advantages

1. Reduced Comparisons
  - Naive method:  $2(n - 1)$  comparisons
  - Divide & Conquer: approximately  $\frac{3n}{2} - 2$  comparisons
  - Hence, more efficient.
2. Better Performance for Large Inputs
  - Works efficiently for large datasets due to reduced operations.
3. Parallel Processing
  - Subarrays can be processed independently → suitable for parallel computing.
4. Structured Approach
  - Clear recursive breakdown makes it easier to analyze.

## Limitations

### 1. Recursive Overhead

- Function calls increase stack usage.

### 2. Not Ideal for Small Inputs

- For small arrays, simple linear scan is faster due to less overhead.

### 3. Extra Space Usage

- Requires additional space due to recursion (call stack).

### 4. Complex Implementation

- Slightly more complex than iterative approach.

## Applications

### 1. Statistical Analysis

- Finding minimum and maximum values in datasets.

### 2. Data Mining

- Used in preprocessing to identify range of data.

### 3. Computer Graphics

- Determining bounding boxes (min/max coordinates).

### 4. Scheduling Problems

- Finding earliest (min) and latest (max) times.

### 5. Optimization Problems

- Helps in identifying extreme values in optimization algorithms.

### 6. Financial Analysis

- Stock price analysis (min/max values over time).

## SELECTION SORT

Selection sort is a simple sorting algorithm. This sorting algorithm, like insertion sort, is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty, and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundaries by one element to the right.

This algorithm is not suitable for large data sets as its average and worst-case complexities are of  $O(n^2)$ , where  $n$  is the number of items.

This type of sorting is called Selection Sort as it works by repeatedly sorting elements. That is: we first find the smallest value in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element



in the second position, and we continue the process in this way until the entire array is sorted.

Pseudo Code:

Algorithm: Selection-Sort (A)

for  $i \leftarrow 1$  to  $n-1$  do

$\min j \leftarrow i$ ;

$\min x \leftarrow A[i]$

    for  $j \leftarrow i + 1$  to  $n$  do

        if  $A[j] < \min x$  then

$\min j \leftarrow j$

$\min x \leftarrow A[j]$

$A[\min j] \leftarrow A[i]$

$A[i] \leftarrow \min x$

### Analysis

Selection sort is among the simplest of sorting techniques and it works very well for small files. It has a quite important application as each item is actually moved at the most once.

Section sort is a method of choice for sorting files with very large objects (records) and small keys. The worst case occurs if the array is already sorted in a descending order and we want to sort them in an ascending order.

Nonetheless, the time required by selection sort algorithm is not very sensitive to the original order of the array to be sorted: the test if  $A[j] < \min x$  is executed the same number of times in every case.

For each  $i$  from  $1$  to  $n - 1$ , there is one exchange and  $n - i$  comparisons, so there is a total of  $n - 1$  exchanges and  $(n - 1) + (n - 2) + \dots + 2 + 1 = n(n - 1)/2$  comparisons.

These observations hold, no matter what the input data is. In the worst case, this could be quadratic, but in the average case, this quantity is  $O(n \log n)$ . It implies that the **running time of Selection sort is quite insensitive to the input.**

**Example:**

Sort the following array in **ascending order** using **Selection Sort**:

**A = [30, 10, 40, 20, 50]**

**Step-by-Step Solution****Initial Array**

[30, 10, 40, 20, 50]

**Pass 1 (i = 0)**

- Find minimum from index 0 to 4 → **10**
- Swap with first element (30)

**Array:**

[10, 30, 40, 20, 50]

**Pass 2 (i = 1)**

- Find minimum from index 1 to 4 → **20**
- Swap with 30

**Array:**

[10, 20, 40, 30, 50]

**Pass 3 (i = 2)**

- Find minimum from index 2 to 4 → **30**
- Swap with 40

**Array:**

[10, 20, 30, 40, 50]

**Pass 4 (i = 3)**

- Find minimum from index 3 to 4 → **40**
- No change needed

**Array:**

[10, 20, 30, 40, 50]

**Final Sorted Array**

[10, 20, 30, 40, 50]

## Advantages

1. **Simple and Easy to Understand**
  - Very easy to implement compared to other sorting algorithms.
2. **In-place Sorting**
  - Requires **no extra memory** (Space Complexity:  $O(1)$ ).
3. **Minimum Number of Swaps**
  - Performs at most  **$(n - 1)$  swaps**, useful when swapping is costly.
4. **Works Well for Small Datasets**
  - Efficient when the input size is small.
5. **No Additional Overhead**
  - Unlike recursive algorithms, no function call overhead.

## Disadvantages

1. **Poor Time Complexity**
  - Time Complexity =  **$O(n^2)$**  in all cases (best, average, worst).
2. **Not Adaptive**
  - Does not take advantage of already sorted data.
3. **Inefficient for Large Datasets**
  - Very slow compared to algorithms like Quick Sort or Merge Sort.
4. **Unstable Sorting Algorithm**
  - May change the relative order of equal elements.
5. **High Number of Comparisons**
  - Always performs  **$n(n-1)/2$  comparisons**, regardless of input.

## Applications

1. **Small Data Sets**
  - Suitable for sorting small lists where simplicity matters.
2. **Memory-Constrained Systems**
  - Used in embedded systems due to **low memory usage**.
3. **When Swapping Cost is High**
  - Preferred when write operations are expensive (e.g., EEPROM memory).
4. **Educational Purposes**
  - Used to teach basic sorting concepts and algorithm design.
5. **Partial Sorting**
  - Useful when only a few smallest elements are needed.

YBKR

YBKR