



EXPERIMENT - 3

Algorithms Design and Analysis Lab

Aim

To implement divide and conquer techniques and analyse its time complexity.

Syeda Reeha Quasar

14114802719

4C7

Experiment – 3

Aim:

To implement divide and conquer techniques and analyse its time complexity.

Theory:

A **divide and conquer algorithm** is a strategy of solving a large problem by

1. breaking the problem into smaller sub-problems
2. solving the sub-problems, and
3. combining them to get the desired output.

Here are the steps involved to perform divide and conquer techniques:

1. **Divide:** Divide the given problem into sub-problems using recursion.
2. **Conquer:** Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
3. **Combine:** Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Time Complexity

The complexity of the divide and conquer algorithm is calculated using the master theorem.

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

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed to have the same size.

$f(n)$ = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

Let us take an example to find the time complexity of a recursive problem.

For a merge sort, the equation can be written as:

$$\begin{aligned}T(n) &= aT(n/b) + f(n) \\ &= 2T(n/2) + O(n)\end{aligned}$$

Where,

$a = 2$ (each time, a problem is divided into 2 subproblems)

$n/b = n/2$ (size of each sub problem is half of the input)

$f(n)$ = time taken to divide the problem and merging the subproblems

$T(n/2) = O(n \log n)$ (To understand this, please refer to the master theorem.)

Now, $T(n) = 2T(n \log n) + O(n)$

$$\approx O(n \log n)$$

Advantages of Divide and Conquer Algorithm

- The complexity for the multiplication of two matrices using the naive method is $O(n^3)$, whereas using the divide and conquer approach (i.e. Strassen's matrix multiplication) is $O(n^{2.8074})$. This approach also simplifies other problems, such as the Tower of Hanoi.
- This approach is suitable for multiprocessing systems.
- It makes efficient use of memory caches.

3.1 Merge Sort:

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

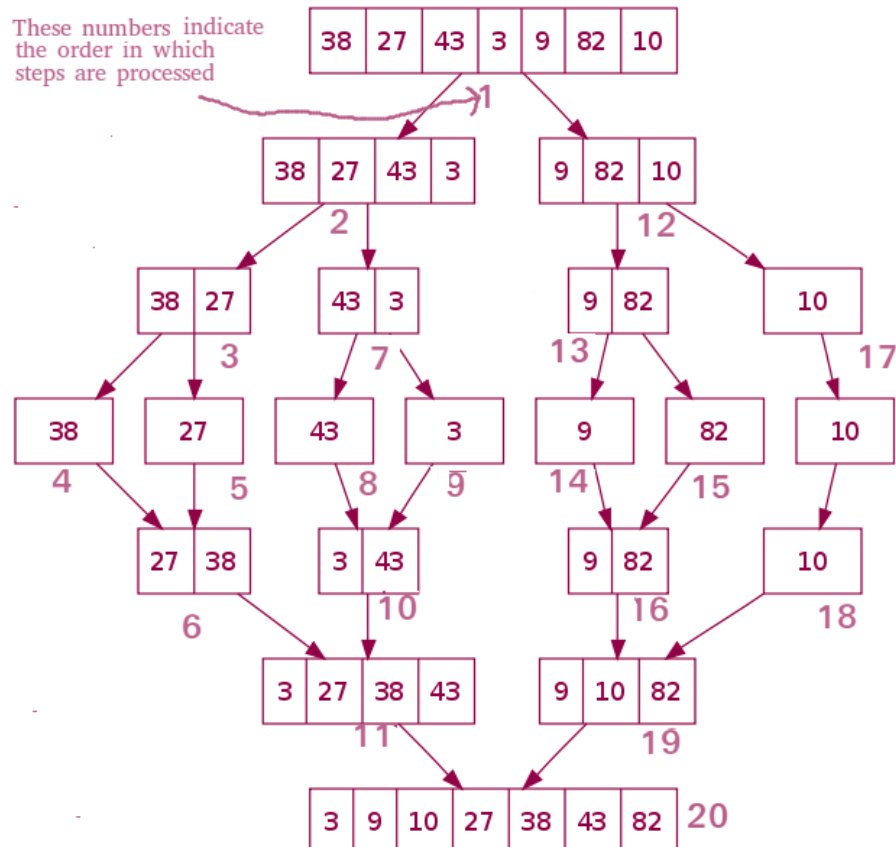
Pseudo code

MergeSort(arr[], l, r)

If $r > l$

1. Find the middle point to divide the array into two halves:
middle $m = l + (r-l)/2$
2. Call mergeSort for first half:
Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
Call merge(arr, l, m, r)

Example:



Result and Analysis

The unordered list of elements gets sorted but additional space is used for merging.

The list of size N is divided into a max of $\log N$ parts, and the merging of all sub lists into a single list takes $O(N)$ time. Time complexity of Merge Sort is $O(n \log n)$ in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves. The recurrence equation used is: $T(n) = 2T(n/2) + O(n)$

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

// merging 2 arrays
void merge(int arr[], int const left, int const midIdx, int const right) {
    int temp[right - left + 1]; // sorted Arr

    int start = left, mid = midIdx + 1, tempIdx = 0;

    // sorted merging of left and right arrays
    while (start <= midIdx && mid <= right) {
        if (arr[start] <= arr[mid]) {
            temp[tempIdx] = arr[start];
            tempIdx++;
            start++;
        }
        else {
            temp[tempIdx] = arr[mid];
            tempIdx++;
            mid++;
        }
    }

    // check and add for remaining element in left arr
    while (start <= midIdx) {
        temp[tempIdx] = arr[start];
        tempIdx++;
    }
}
```

```
        start++;
    }

    // check and add for remaining element in right arr
    while (mid <= right) {
        temp[tempIdx] = arr[mid];
        tempIdx++;
        mid++;
    }

    for (int i = left; i <= right; i++) {
        arr[i] = temp[i - left];
    }
}

void mergeSort(int array[], int const begin, int const end) {
    if (begin >= end) {
        return; // Returns recursively
    }
    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

void printArr(int arr[], int size) {
    for (auto i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << "\n\n" << endl;
}

int main() {
    int n = rand() % 100;
    int arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }
    cout << "Given array: \n";
    printArr(arr, n);

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);
```

```
mergeSort(arr, 0, n - 1);

auto end = chrono::high_resolution_clock::now();

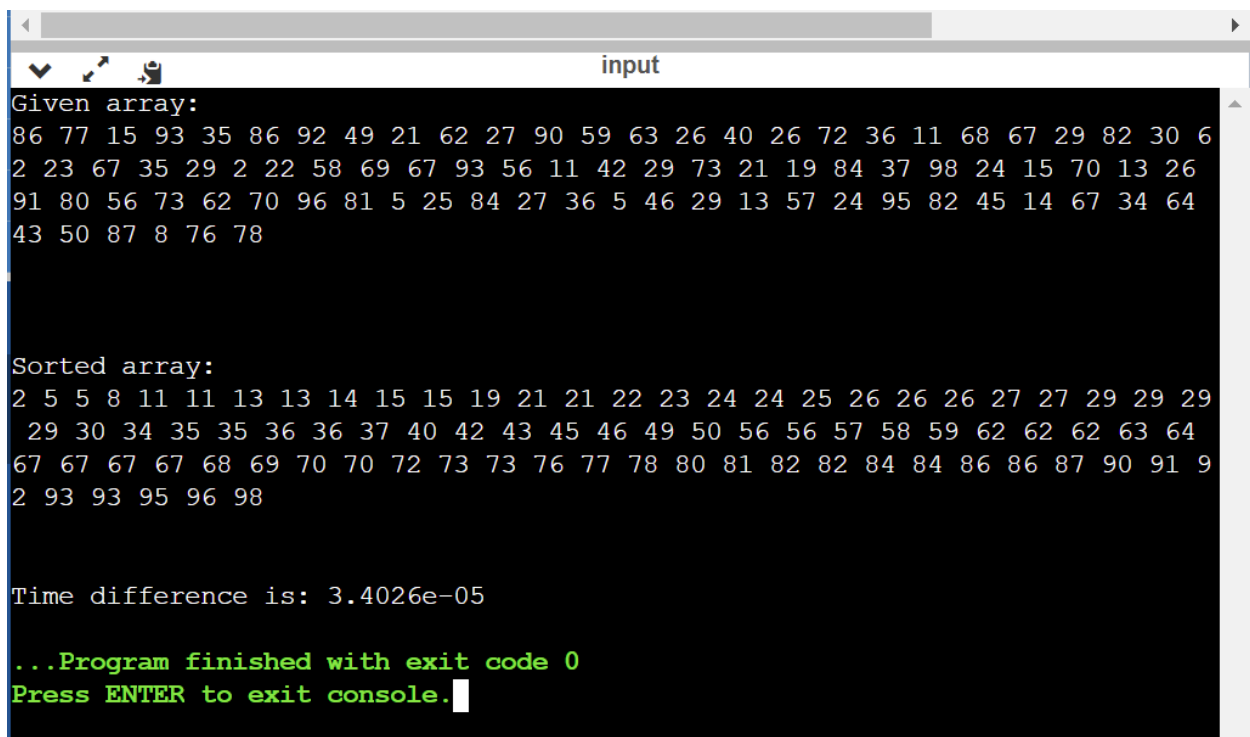
cout << "\nSorted array: \n";
printArr(arr, n);

double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
time_taken *= 1e-9;

cout << "Time difference is: " << time_taken << setprecision(6);

return 0;
}
```

Output:



```
input
Given array:
86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 6
2 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70 13 26
91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64
43 50 87 8 76 78

Sorted array:
2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 27 27 29 29 29
29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 62 62 62 63 64
67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86 86 87 90 91 9
2 93 93 95 96 98

Time difference is: 3.4026e-05

...Program finished with exit code 0
Press ENTER to exit console.
```

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

// merging 2 arrays
void merge(int arr[], int const left, int const mid, int const right) {
    auto const subArr1 = mid - left + 1;
    auto const subArr2 = right - mid;

    auto *leftArr = new int[subArr1],
        *rightArr = new int[subArr2];

    for (auto i = 0; i < subArr1; i++)
        leftArr[i] = arr[left + i];

    for (auto j = 0; j < subArr2; j++)
        rightArr[j] = arr[mid + 1 + j];

    auto indexOfSubArr1 = 0,
        indexOfSubArr2 = 0;
    int indexOfMergedArr = left;

    // sorted merging of left and right arrays
    while (indexOfSubArr1 < subArr1 && indexOfSubArr2 < subArr2) {
        if (leftArr[indexOfSubArr1] <= rightArr[indexOfSubArr2]) {
            arr[indexOfMergedArr] = leftArr[indexOfSubArr1];
            indexOfSubArr1++;
        }
        else {
            arr[indexOfMergedArr] = rightArr[indexOfSubArr2];
            indexOfSubArr2++;
        }
        indexOfMergedArr++;
    }

    // check and add for remaining element in left arr
    while (indexOfSubArr1 < subArr1) {
        arr[indexOfMergedArr] = leftArr[indexOfSubArr1];
        indexOfSubArr1++;
        indexOfMergedArr++;
    }
}
```



```
        // check and add for remaining element in left arr
        while (indexOfSubArr2 < subArr2) {
            arr[indexOfMergedArr] = rightArr[indexOfSubArr2];
            indexOfSubArr2++;
            indexOfMergedArr++;
        }
    }

void mergeSort(int array[], int const begin, int const end) {
    if (begin >= end) {
        return; // Returns recursively
    }
    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

void printArr(int arr[], int size) {
    for (auto i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << "\n\n" << endl;
}

int main() {
    int n = rand() % 100;
    int arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }
    cout << "Given array: \n";
    printArr(arr, n);

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    mergeSort(arr, 0, n - 1);

    auto end = chrono::high_resolution_clock::now();

    cout << "\nSorted array: \n";
```

```

    printArr(arr, n);

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);

    return 0;
}

```

Output:

```

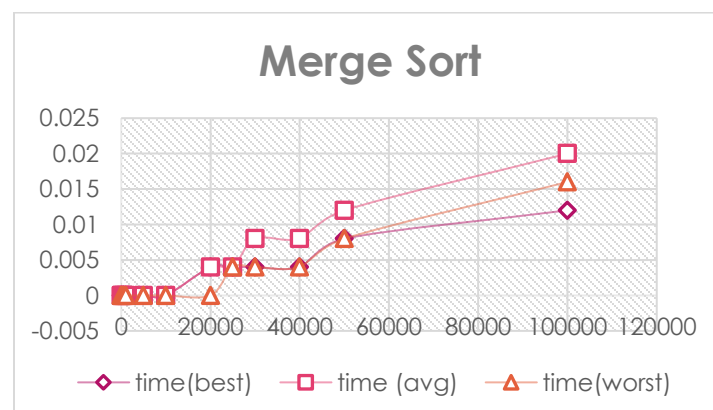
input
Given array:
86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 6
2 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70 13 26
91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64
43 50 87 8 76 78

Sorted array:
2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 27 27 29 29 29
29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 62 62 62 63 64
67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86 86 87 90 91 9
2 93 93 95 96 98

Time difference is: 5.0381e-05

...Program finished with exit code 0
Press ENTER to exit console.

```



Batch Analysis:

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

// merging 2 arrays
void merge(int arr[], int const left, int const mid, int const right) {
    auto const subArr1 = mid - left + 1;
    auto const subArr2 = right - mid;

    auto *leftArr = new int[subArr1],
        *rightArr = new int[subArr2];

    for (auto i = 0; i < subArr1; i++)
        leftArr[i] = arr[left + i];

    for (auto j = 0; j < subArr2; j++)
        rightArr[j] = arr[mid + 1 + j];

    auto indexOfSubArr1 = 0,
        indexOfSubArr2 = 0;
    int indexOfMergedArr = left;

    // sorted merging of left and right arrays
    while (indexOfSubArr1 < subArr1 && indexOfSubArr2 < subArr2) {
        if (leftArr[indexOfSubArr1] <= rightArr[indexOfSubArr2]) {
            arr[indexOfMergedArr] = leftArr[indexOfSubArr1];
            indexOfSubArr1++;
        }
        else {
            arr[indexOfMergedArr] = rightArr[indexOfSubArr2];
            indexOfSubArr2++;
        }
        indexOfMergedArr++;
    }

    // check and add for remaining element in left arr
    while (indexOfSubArr1 < subArr1) {
        arr[indexOfMergedArr] = leftArr[indexOfSubArr1];
        indexOfSubArr1++;
    }
```

```
        indexOfMergedArr++;
    }

    // check and add for remaining element in left arr
    while (indexOfSubArr2 < subArr2) {
        arr[indexOfMergedArr] = rightArr[indexOfSubArr2];
        indexOfSubArr2++;
        indexOfMergedArr++;
    }
}

void mergeSort(int array[], int const begin, int const end) {
    if (begin >= end) {
        return; // Returns recursively
    }
    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

double sortApp(int n) {
    // int n = rand() % 100;
    int arr[n];
```

```
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }
// cout << "Given array: \n";
// printArr(arr, n);

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    mergeSort(arr, 0, n - 1);

    auto end = chrono::high_resolution_clock::now();

// cout << "\nSorted array: \n";
// printArr(arr, n);

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    // cout << "Time difference is: " << time_taken << setprecision(6);

    return time_taken;
}

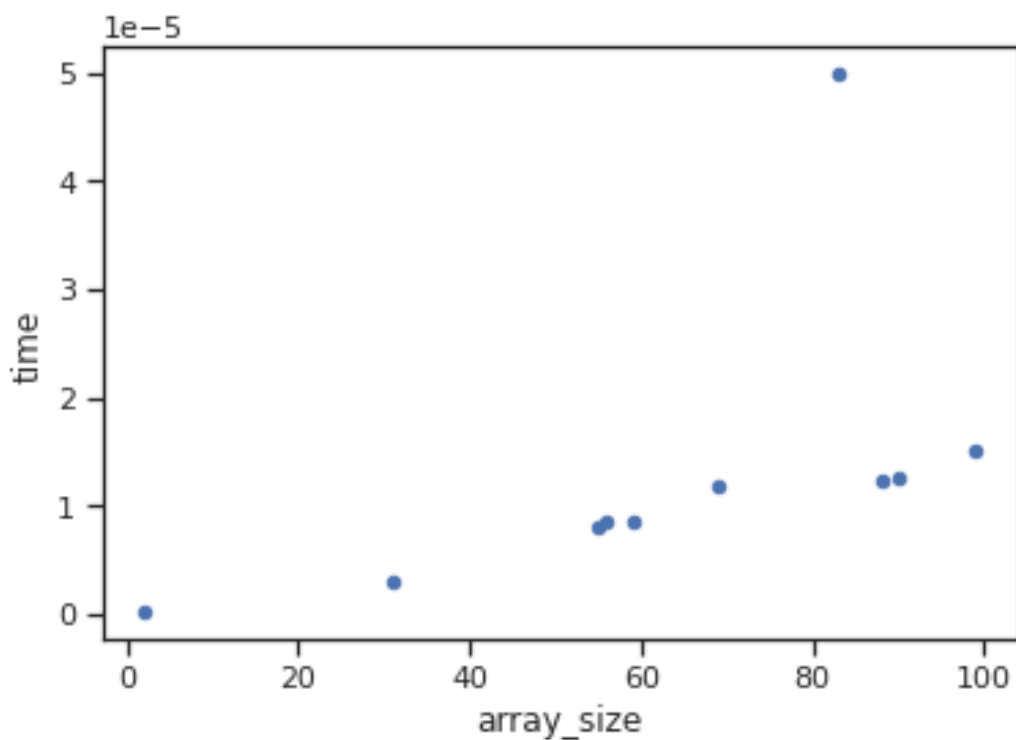
int main() {
    double times[10];
    int ns[10];
    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
        ns[x] = n;
        times[x] = sortApp(n);
    }
    cout << "value of n's: " << endl;
    printArray(ns, 10);
    cout << "time for each n: " << endl;
    printArray(times, 10);
}
```

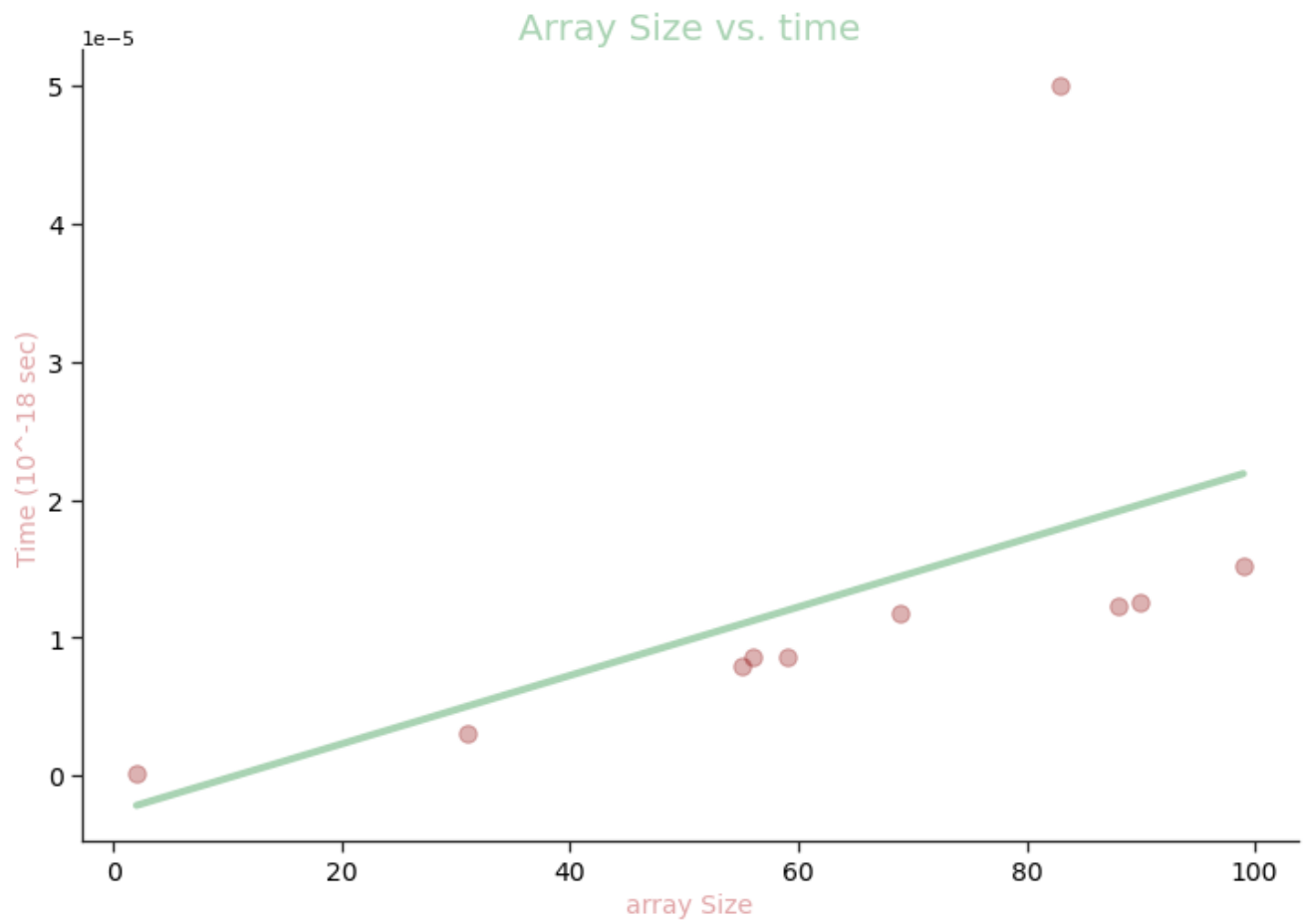
Output:

```
input
value of n's:
83, 88, 56, 69, 90, 59, 2, 99, 55, 31,
time for each n:
7.1799e-05, 2.4215e-05, 1.657e-05, 2.1988e-05, 2.472e-05, 1.6775e-05, 2.34e-
07, 2.9243e-05, 1.5824e-05, 6.567e-06,

...Program finished with exit code 0
Press ENTER to exit console.
```

```
arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]
time = [5.0004e-05, 1.2248e-05, 8.572e-06, 1.1836e-05, 1.2539e-05, 8.589e-
06, 1.34e-07, 1.5209e-05, 7.996e-06, 3.056e-06]
```





Merge Sort

Viva Questions

1. Why is time complexity of merge sort?

Ans.

$$T(n) = 2T(n/2) + \theta(n)$$

Time complexity of Merge Sort is $\theta(n \log n)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

$$O(n \log n)$$

Auxiliary Space: $O(n)$

Algorithmic Paradigm: Divide and Conquer

Sorting In Place: No in a typical implementation

Stable: Yes

2. Is it possible to do merge sort in place?

Ans.

There do exist in-place merge sorts. They must be implemented carefully. First, naive in-place merge such as described here isn't the right solution. It downgrades the performance to $O(N^2)$.

3. What are the total number of passes in merge sort of n numbers?

Ans.

$$\log(n)$$

Suppose an array of n elements.

Now iterate through this array just one time. The time complexity of this operation is $O(n)$. This time complexity tells us that we are iterating one time through an array of length n.

Now iterate through this array n times. The time complexity of this operation is $O(n*n)$. This time complexity tells us that we are iterating n times through an array of length n .

Now the time complexity of merge sort is $O(n\log(n))$. This says that we are iterating through an n element array, $\log(n)$ times.

4. Given two sorted lists of size m , n then what are the number of comparisons needed in the worst case by merge sort?

Ans.

To merge two lists of size m and n , we need to do **$m+n-1$ comparisons** in worst case.

5. What is the output of merge sort after the 2nd pass given the following sequence of numbers: 3,41,52,26,38,57,9,49

Ans.

3,26,41,52,9,38,49,57

6. Give any application of merge sort?

Ans.

- Merge Sort is useful for sorting linked lists in $O(n \log n)$ time
- Merge sort can be implemented without extra space for linked lists
- Merge sort is used for counting inversions in a list
- Merge sort is used in external sorting

3.2 Quick Sort:

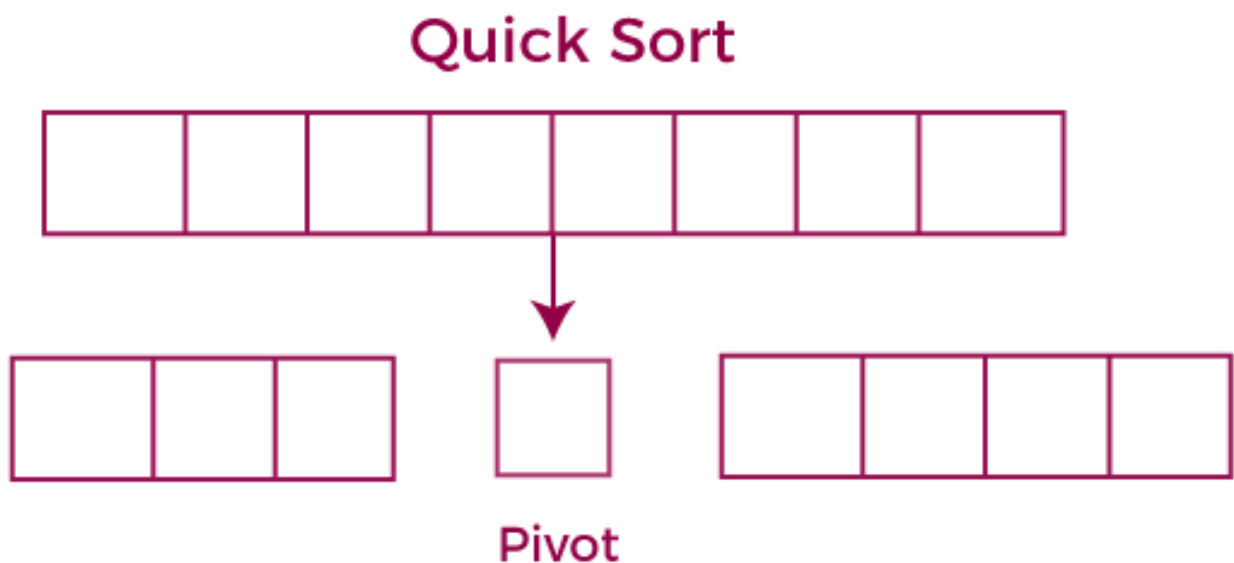
Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **$n \log n$** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot. After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element or the leftmost element of the given array.
- Select median as the pivot element.

Pseudo code

Algorithm:

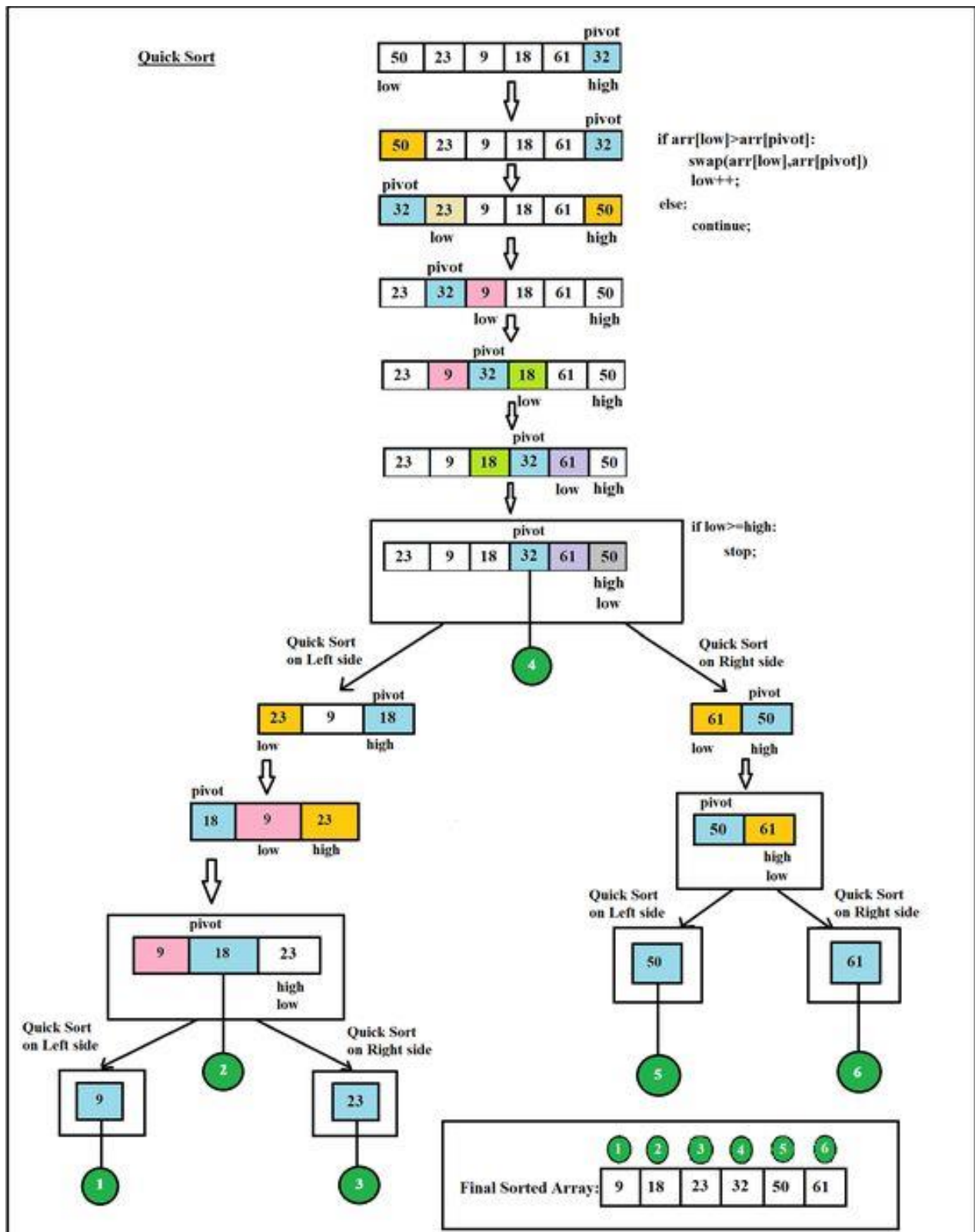
```
1. QUICKSORT (array A, start, end)
2. {
3.   1 if (start < end)
4.   2 {
5.   3 p = partition(A, start, end)
6.   4 QUICKSORT (A, start, p - 1)
7.   5 QUICKSORT (A, p + 1, end)
8.   6 }
9. }
```

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in a place.

```
1. PARTITION (array A, start, end)
2. {
3.   1 pivot ? A[end]
4.   2 i ? start-1
5.   3 for j ? start to end -1 {
6.   4 do if (A[j] < pivot) {
7.   5 then i ? i + 1
8.   6 swap A[i] with A[j]
9.   7 }}
10.   8 swap A[i+1] with A[end]
11.   9 return i+1
12. }
```

Example:



Result and Analysis

Time complexity

Case	Time Complexity
Best Case	$O(n \cdot \log n)$
Average Case	$O(n \cdot \log n)$
Worst Case	$O(n^2)$

- **Best Case Complexity** - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **$O(n \cdot \log n)$** .
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **$O(n \cdot \log n)$** .
- **Worst Case Complexity** - In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$** .

Space Complexity

Space Complexity	$O(n \cdot \log n)$
Stable	NO

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

// A utility function to swap two elements
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// last element pivot and lower in 1 half and greater in 1 half divide
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);    // Index of smaller element and indicates the right
    position of pivot found so far

    for (int j = low; j <= high - 1; j++)
    {
        // j == current element
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int p = partition(arr, low, high);

        // Separately sort elements before partition and after partition
```

```
        quickSort(arr, low, p - 1);
        quickSort(arr, p + 1, high);
    }
}

void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << "\n\n" << endl;
}

int main()
{
    int n = rand() % 100;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }
    cout << "Given array: \n";
    printArray(arr, n);

    auto start = chrono::high_resolution_clock::now();

    quickSort(arr, 0, n - 1);

    auto end = chrono::high_resolution_clock::now();

    cout << "\nSorted array: \n";
    printArray(arr, n);

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);

    return 0;
}
```

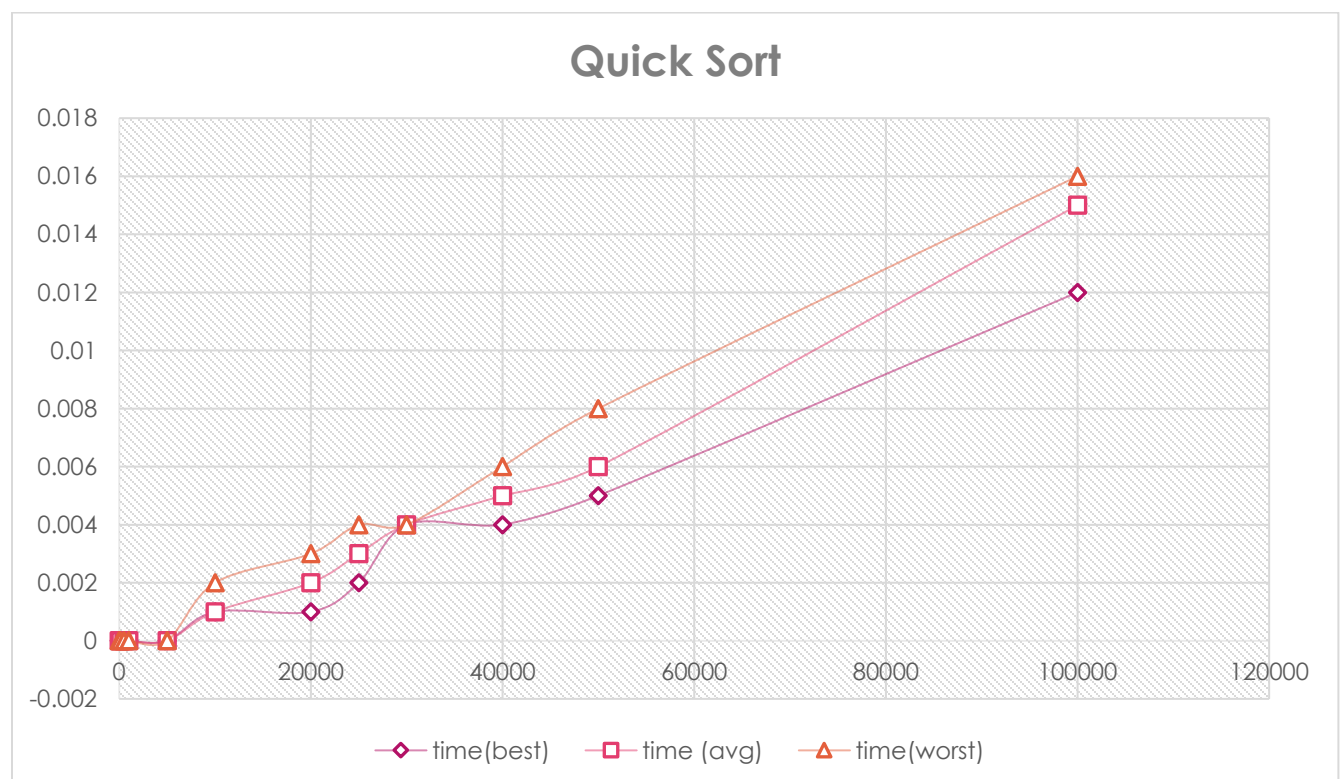
Output:

```
input
Given array:
86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 6
2 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70 13 26
91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64
43 50 87 8 76 78

Sorted array:
2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 27 27 29 29 29
29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 62 62 62 63 64
67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86 86 87 90 91 9
2 93 93 95 96 98

Time difference is: 3.4026e-05

...Program finished with exit code 0
Press ENTER to exit console.
```



Batch Analysis:

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

// A utility function to swap two elements
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// last element pivot and lower in 1 half and greater in 1 half divide
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);    // Index of smaller element and indicates the right
    position of pivot found so far

    for (int j = low; j <= high - 1; j++)
    {
        // j == current element
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int p = partition(arr, low, high);

        // Separately sort elements before partition and after partition
```

```
        quickSort(arr, low, p - 1);
        quickSort(arr, p + 1, high);
    }
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

double sortApp(int n) {
    int arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    quickSort(arr, 0, n - 1);

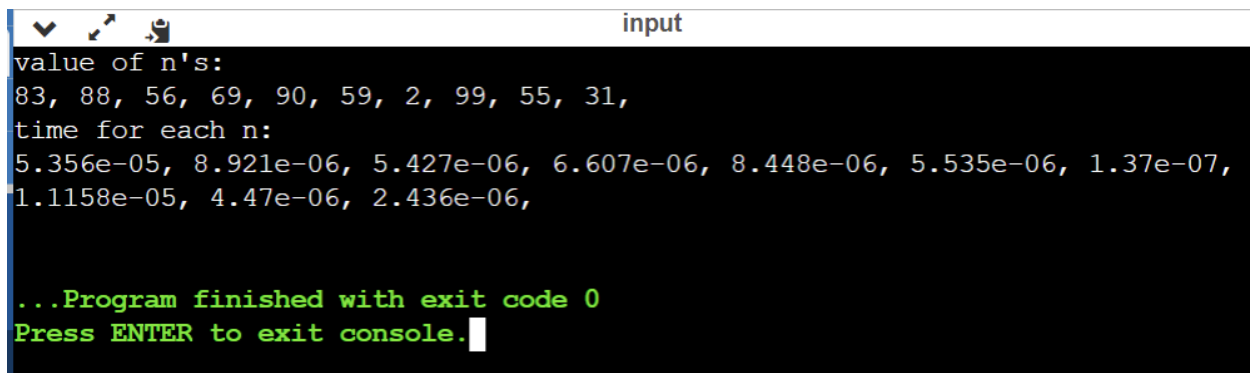
    auto end = chrono::high_resolution_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    return time_taken;
}
```

```
int main() {
    double times[10];
    int ns[10];
    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
        ns[x] = n;
        times[x] = sortApp(n);
    }
    cout << "value of n's: " << endl;
    printArray(ns, 10);
    cout << "time for each n: " << endl;
    printArray(times, 10);
}
```

Output:

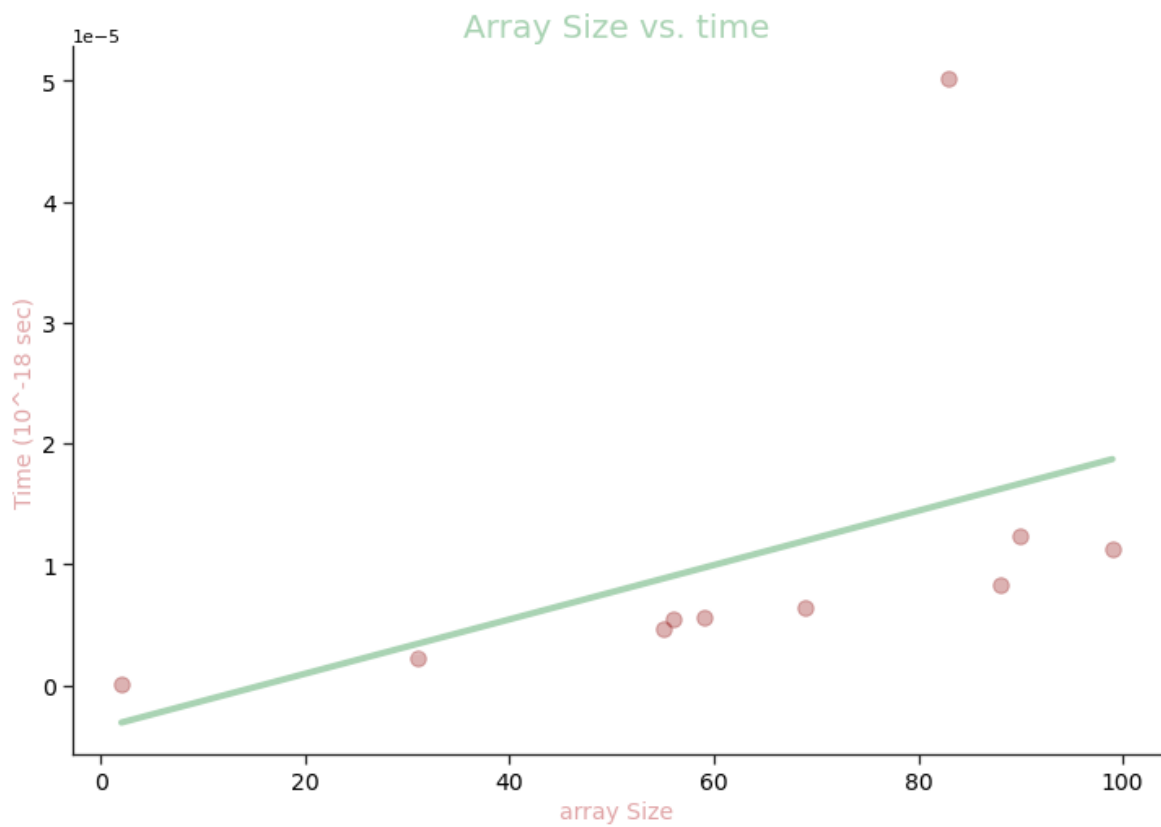
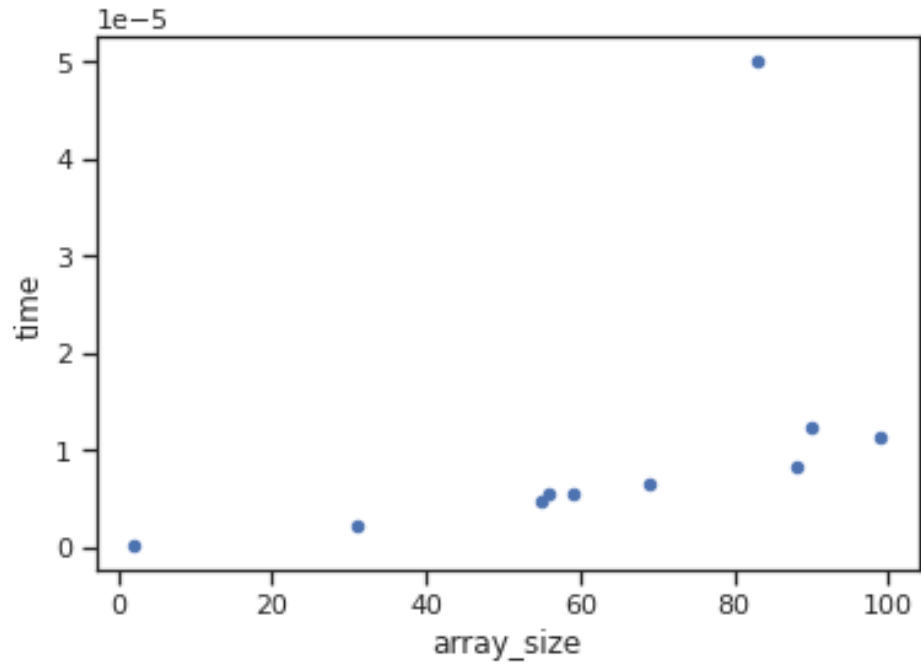


```
input
value of n's:
83, 88, 56, 69, 90, 59, 2, 99, 55, 31,
time for each n:
5.356e-05, 8.921e-06, 5.427e-06, 6.607e-06, 8.448e-06, 5.535e-06, 1.37e-07,
1.1158e-05, 4.47e-06, 2.436e-06,

...Program finished with exit code 0
Press ENTER to exit console.
```

```
arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]
```

```
time = [5.017e-05, 8.247e-06, 5.499e-06, 6.48e-06, 1.2334e-05, 5.57e-06,
1.35e-07, 1.1314e-05, 4.678e-06, 2.284e-06]
```



Quick Sort

Viva Questions

1. What value of q does PARTITION return when all elements in the array $A[p \dots r]$ have the same value?

Ans.

If all elements are equal, then when PARTITION returns, $q = r$ and all elements in $A[p \dots q-1]$ are equal. We get the recurrence $T(n) = T(n-1) + \Theta(n)$ for the running time, and so $T(n) = \Theta(n^2)$.

PARTITION(A, p, r)
 1 $x = A[r]$ 2 $i = p - 1$ 3 for $j = p$ to $r - 1$ 4 if $A[j] \leq x$ 5 $i = i + 1$ 6 exchange $A[i]$ with $A[j]$ 7 exchange $A[i + 1]$ with $A[r]$ 8 return $i + 1$
 Since the if test on the analogue of line 4 is thus successful when all entries are equal, in that case the value of k when the for loop terminates is $k = r-1$, so the value of q returned is $q = k+1 = r$.

2. Give a brief argument that the running time of PARTITION on a sub array of size n is $\Theta(n)$.

Ans.

There is a for statement whose body executes $r - 1 - p = \Theta(n)$ times. In the worst case every time the body of the if is executed, but it takes constant time and so does the code outside of the loop. Thus the running time is $\Theta(n)$.

3. How would you modify QUICKSORT to sort in non-increasing order?

Ans.

It is one of the fastest sorting algorithms known and is the method of choice in most sorting libraries. QUICKSORT is based on the divide and conquer strategy.

4. Why Quick sort is considered as best sorting?

Ans.

The cache efficiency argument has already been explained in detail. In addition, there is an intrinsic argument, why Quicksort is fast. If implemented like with two "crossing pointers", e.g. here, the inner loops have a very small body. As this is the code executed most often, this pays off.

5. What is randomized version of Quick sort?

Ans.

Randomized quick sort chooses a random element as a pivot. It is done so as to avoid the worst case of quick sort in which the input array is already sorted.

3.3 Matrix Multiplication and Strassen's Algorithm

Given a sequence of matrices A_1, A_2, \dots, A_n , insert parentheses so that the product of the matrices, in order, is unambiguous and needs the minimal number of multiplications

Matrix multiplication is associative: $A_1 (A_2 A_3) = (A_1 A_2) A_3$

A product is unambiguous if no factor is multiplied on both the left and the right and all factors are either a single matrix or an unambiguous product.

Multiplying an $i \times j$ and a $j \times k$ matrix requires ijk multiplications. Each element of the product requires j multiplications, and there are ik elements

$$\begin{array}{ccc}
 & \vec{b}_1 & \vec{b}_2 \\
 & \downarrow & \downarrow \\
 \begin{array}{l} \vec{a}_1 \rightarrow \\ \vec{a}_2 \rightarrow \end{array} & \begin{bmatrix} 1 & 7 \\ 2 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & 3 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} \vec{a}_1 \cdot \vec{b}_1 & \vec{a}_1 \cdot \vec{b}_2 \\ \vec{a}_2 \cdot \vec{b}_1 & \vec{a}_2 \cdot \vec{b}_2 \end{bmatrix} \\
 A & B & C
 \end{array}$$

Number of Parenthesizations

Given the matrices A_1, A_2, A_3, A_4 . Assume the dimensions of $A_1 = d_0 \times d_1$, etc. Below are the five possible parenthesizations of these arrays, along with the number of multiplications:

1. $(A_1 A_2)(A_3 A_4): d_0 d_1 d_2 + d_2 d_3 d_4 + d_0 d_2 d_4$
2. $((A_1 A_2) A_3) A_4: d_0 d_1 d_2 + d_0 d_2 d_3 + d_0 d_3 d_4$
3. $(A_1 (A_2 A_3)) A_4: d_1 d_2 d_3 + d_0 d_1 d_3 + d_0 d_3 d_4$
4. $A_1 ((A_2 A_3) A_4): d_1 d_2 d_3 + d_1 d_3 d_4 + d_0 d_1 d_4$

$$5. \quad A_1(A_2(A_3A_4)):d_2d_3d_4+d_1d_2d_4+d_0d_1d_4$$

The number of parenthesizations is at least $T(n) \geq T(n-1) + T(n-1)$. Since the number with the first element removed is $T(n-1)$, which is also the number with the last removed. Thus the number of parenthesizations is $\Omega(2^n)$. The number is actually $T(n) = \sum_{k=1}^{n-1} T(k)T(n-k)$ which is related to the *Catalan numbers*. This is because the original product can be split into 2 sub products in k places. Each split is to be parenthesized optimally. This recurrence is related to the *Catalan numbers*.

Characterizing the Optimal Parenthesization

An optimal parenthesization of $A_1 \dots A_n$ must break the product into two expressions, each of which is parenthesized or is a single array. Assume the break occurs at position k . In the optimal solution, the solution to the product $A_1 \dots A_k$ must be optimal otherwise, we could improve $A_1 \dots A_n$ by improving $A_1 \dots A_k$ but the solution to $A_1 \dots A_n$ is known to be optimal. This is a contradiction. Thus the solution to $A_1 \dots A_n$ is known to be optimal.

Principle of Optimality

This problem exhibits the Principle of Optimality. The optimal solution to product $A_1 \dots A_n$ contains the optimal solution to two sub products. Thus we can use Dynamic Programming. Consider a recursive solution

Then improve its performance with memoization or by rewriting bottom up

Matrix Dimensions

Consider matrix product $A_1 \times \dots \times A_n$. Let the dimensions of matrix A_i be $d_{i-1} \times d_i$. Thus the dimensions of matrix product $A_i \times \dots \times A_j$ are $d_{i-1} \times d_j$.

Recursive Solution

Let $M[i, j]$ represent the number of multiplications required for matrix product $A_i \times \dots \times A_j$. For $1 \leq i \leq j < n$: $M[i, i] = 0$ since no product is required

The optimal solution of $A_i \times A_j$ must break at some point, k , with $i \leq k < j$

Thus, $M[i,j] = M[i,k] + M[k+1,j] + d_{i-1}d_kd_j$

Thus, $M[i,j] = \begin{cases} 0 & \text{if } i=j \\ \min_{i \leq k < j} \{M[i,k] + M[k+1,j] + d_{i-1}d_kd_j\} & \text{if } i < j \end{cases}$

Pseudo code:

// Matrix A_i has dimension $p[i-1] \times p[i]$ for $i = 1..n$

MATRIX-CHAIN-ORDER (p)

$n \leftarrow \text{length}[p] - 1$

for $i \leftarrow 1$ to n

do $m[i,i] \leftarrow 0$

for $l \leftarrow 2$ to n

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

do $j \leftarrow i+l-1$

$m[i,j] \leftarrow \infty$

for $k \leftarrow i$ to $j-1$

do $q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$

if $q < m[i,j]$

then $m[i,j] \leftarrow q$

$s[i,j] \leftarrow k$

return m and s

PRINT-OPTIMAL-PARENS (s, i, j)

if $i=j$

then print " A_i "

else print " ("

PRINT-OPTIMAL-PARENS ($s, i, s[i,j]$)

PRINT-OPTIMAL-PARENS ($s, s[i,j]+1, j$)

An optimal solution can be constructed from the computed information stored in the table $s[1...n, 1...n]$. The earlier matrix multiplication can be computed recursively.

$$\begin{aligned}
 p1 &= a(f - h) & p2 &= (a + b)h \\
 p3 &= (c + d)e & p4 &= d(g - e) \\
 p5 &= (a + d)(e + h) & p6 &= (b - d)(g + h) \\
 p7 &= (a - c)(e + f)
 \end{aligned}$$

The $A \times B$ can be calculated using above seven multiplications.

Following are values of four sub-matrices of result C

$$\begin{array}{c} \left[\begin{array}{c|c} a & b \\ \hline c & d \end{array} \right] \end{array} \times \begin{array}{c} \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array} \right] \end{array} = \begin{array}{c} \left[\begin{array}{c|c} p5 + p4 - p2 + p6 & p1 + p2 \\ \hline p3 + p4 & p1 + p5 - p3 - p7 \end{array} \right] \end{array}$$

A
 B
 C

A , B and C are square matrices of size $N \times N$

a , b , c and d are submatrices of A , of size $N/2 \times N/2$

e , f , g and h are submatrices of B , of size $N/2 \times N/2$

$p1$, $p2$, $p3$, $p4$, $p5$, $p6$ and $p7$ are submatrices of size $N/2 \times N/2$

Result and Analysis

Clearly, the space complexity of this procedure is $O(n^2)$. Since the tables m and s require $O(n^2)$ space. As far as the time complexity is concern, a simple inspection of the for-loop(s) structures gives us a running time of the procedure. Since, the three for-loops are nested three deep, and each one of them iterates at most n times (that is to say indices L , i , and j takes on at most $n - 1$ values). Therefore, the running time of this procedure is $O(n^3)$.

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

$$T(N) = 7T(N/2) + O(N^2)$$

From [Master's Theorem](#), time complexity of this method is

$$O(N^{\log_2 7}) \text{ which is approximately } O(N^{2.8074})$$

Matrix Multiplication:

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

void printMatrix(int arr[10][10], int r, int c)
{
    for (int i = 0; i < r; ++i)
        for (int j = 0; j < c; ++j)
        {
            cout << " " << arr[i][j];
            if (j == c - 1)
                cout << endl;
        }
}

int main()
{
    int a[10][10], b[10][10], multipliedMatrix[10][10], r1, c1, r2, c2, i, j, k;

    cout << "Enter rows and columns for first matrix: ";
    cin >> r1 >> c1;
    cout << "Enter rows and columns for second matrix: ";
    cin >> r2 >> c2;

    while (c1 != r2) // matrix multiplication validation check
    {
        cout << "Error! column of first matrix not equal to row of second.";

        cout << "Enter rows and columns for first matrix: ";
        cin >> r1 >> c1;

        cout << "Enter rows and columns for second matrix: ";
        cin >> r2 >> c2;
    }

    cout << endl
        << "\n Enter elements of matrix 1:" << endl;
```

```
for (i = 0; i < r1; ++i)
    for (j = 0; j < c1; ++j)
    {
        cin >> a[i][j];
    }

cout << endl
    << "\n Enter elements of matrix 2:" << endl;
for (i = 0; i < r2; ++i)
    for (j = 0; j < c2; ++j)
    {
        cin >> b[i][j];
    }

// Initializing elements of matrix multipliedMatrix to 0.
for (i = 0; i < r1; ++i)
    for (j = 0; j < c2; ++j)
    {
        multipliedMatrix[i][j] = 0;
    }

auto start = chrono::high_resolution_clock::now();
// unsync the I/O of C and C++.
ios_base::sync_with_stdio(false);


for (i = 0; i < r1; ++i) // multiplication
    for (j = 0; j < c2; ++j)
        for (k = 0; k < c1; ++k)
        {
            multipliedMatrix[i][j] += a[i][k] * b[k][j];
        }

auto end = chrono::high_resolution_clock::now();

cout << "multiply of the matrix=\n";
printMatrix(multipliedMatrix, r1, c2);

double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
time_taken *= 1e-9;
cout << "Time difference is: " << time_taken << setprecision(6) << endl;

return 0;
}
```

Output:

```
Enter rows and columns for first matrix: 2 2
Enter rows and columns for second matrix: 3 3
Error! column of first matrix not equal to row of second.
Enter rows and columns for first matrix: 2 2
Enter rows and columns for second matrix: 2 2

Enter elements of matrix 1:
3 4
2 1

Enter elements of matrix 2:
1 5
3 7
multiply of the matrix=
15 43
5 17
Time difference is: 4.7856e-05

...Program finished with exit code 0
Press ENTER to exit console.
```

```
multiply of the matrix=
15 43
5 17
Time difference is: 4.7856e-05

...Program finished with exit code 0
Press ENTER to exit console.
```

Strassen algorithm:

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

void printMatrix(int arr[2][2], int r, int c)
{
    for (int i = 0; i < r; ++i)
        for (int j = 0; j < c; ++j)
        {
            cout << " " << arr[i][j];
            if (j == c - 1)
                cout << endl;
        }
}

int main()
{
    int a[2][2], b[2][2], mult[2][2];
    int m1, m2, m3, m4, m5, m6, m7, i, j;

    cout << "Matrix Multiplication Strassen's method: \n";
    cout << "Enter the elements of 2x2 Matrix 1:\n";
    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
        {
            cin >> a[i][j];
        }
    }

    cout << "Enter the elements of 2x2 Matrix 2:\n";
    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
        {
            cin >> b[i][j];
        }
    }
}
```

```
}

auto start = chrono::high_resolution_clock::now();
// unsync the I/O of C and C++.
ios_base::sync_with_stdio(false);

m1 = (a[0][0] + a[1][1]) * (b[0][0] + b[1][1]);
m2 = (a[1][0] + a[1][1]) * b[0][0];
m3 = a[0][0] * (b[0][1] - b[1][1]);
m4 = a[1][1] * (b[1][0] - b[0][0]);
m5 = (a[0][0] + a[0][1]) * b[1][1];
m6 = (a[1][0] - a[0][0]) * (b[0][0] + b[0][1]);
m7 = (a[0][1] - a[1][1]) * (b[1][0] + b[1][1]);

mult[0][0] = m1 + m4 - m5 + m7;
mult[0][1] = m3 + m5;
mult[1][0] = m2 + m4;
mult[1][1] = m1 - m2 + m3 + m6;


auto end = chrono::high_resolution_clock::now();

cout << "\nProduct of matrices is: \n";
printMatrix(mult, 2, 2);

double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
time_taken *= 1e-9;

cout << "Time difference is: " << time_taken << setprecision(6) << endl;

return 0;
}
```


Output:


```
Matrix Multiplication Strassen's method:
Enter the elements of 2x2 Matrix 1:
3 4
2 1
Enter the elements of 2x2 Matrix 2:
1 5
3 7

Product of matrices is:
 15 43
  5 17
Time difference is: 7.2175e-05

...Program finished with exit code 0
Press ENTER to exit console.
```

```
Product of matrices is:
 15 43
  5 17
Time difference is: 7.2175e-05

...Program finished with exit code 0
Press ENTER to exit console.
```



```
Matrix Multiplication Strassen's method:
Enter the elements of 2x2 Matrix 1:
2 4
3 6
Enter the elements of 2x2 Matrix 2:
9 3
2 3

Product of matrices is:
 26 18
 39 27
Time difference is: 6.3373e-05

...Program finished with exit code 0
Press ENTER to exit console.
```

Viva Questions

1. What is the recurrence relation for MCM problem?

Ans.

Matrix Chain Multiplication + Dynamic Programming + Recurrence Relation

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

$$(ABC)D = (AB)(CD) = A(BCD) = \dots$$

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10×30 matrix, B is a 30×5 matrix, and C is a 5×60 matrix. Then,

$$(AB)C = (10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500 \text{ operations}$$

$$A(BC) = (30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000 \text{ operations.}$$

Clearly the first parenthesization requires less number of operations.

Given an array $p[]$ which represents the chain of matrices such that the i th matrix A_i is of dimension $p[i-1] \times p[i]$. We need to write a function `MatrixChainOrder()` that should return the minimum number of multiplications needed to multiply the chain.

2. Let A_1, A_2, A_3, A_4, A_5 be five matrices of dimensions $2 \times 3, 3 \times 5, 5 \times 2, 2 \times 4, 4 \times 3$ respectively. Find the minimum number of scalar multiplications required to find the product $A_1 A_2 A_3 A_4 A_5$ using the basic matrix multiplication method?

Ans.

We have many ways to do matrix chain multiplication because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result of the matrix chain multiplication obtained will remain the same. Here we have four matrices A_1, A_2, A_3 , and A_4 , we would have:

$$((A_1 A_2) A_3) A_4 = ((A_1 (A_2 A_3)) A_4) = (A_1 A_2) (A_3 A_4) = A_1 ((A_2 A_3) A_4) = A_1 (A_2 (A_3 A_4)).$$

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. Here, A_1 is a 10×5 matrix, A_2 is a 5×20 matrix, and A_3 is a 20×10 matrix, and A_4 is 10×5 .

If we multiply two matrices A and B of order $l \times m$ and $m \times n$ respectively, then the number of scalar multiplications in the multiplication of A and B will be $l \times m \times n$.

Then,

The number of scalar multiplications required in the following sequence of matrices will be :

$$A_1 ((A_2 A_3) A_4) = (5 \times 20 \times 10) + (5 \times 10 \times 5) + (10 \times 5 \times 5) = 1000 + 250 + 250 = 1500.$$

All other parenthesized options will require number of multiplications more than 1500.

3. Consider the two matrices P and Q which are 10×20 and 20×30 matrices respectively. What is the number of multiplications required to multiply the two matrices?

Ans.

The number of multiplications required is $10 \times 20 \times 30$.

4. Consider the matrices P, Q and R which are 10 x 20, 20 x 30 and 30 x 40 matrices respectively. What is the minimum number of multiplications required to multiply the three matrices?

Ans.

The minimum number of multiplications are 18000. This is the case when the matrices are parenthesized as $(P*Q)*R$.

5. Can this problem be solved by greedy approach. Justify?

Ans.

Unfortunately, **there is no good "greedy choice"** for Matrix Chain Multiplication, meaning that for any choice that's easy to compute, there is always some input sequence of matrices for which your greedy algorithm will not find the optimum parenthesization.

At each step only least value is selected among all elements of in the array $p[x, y]$, so that the multiplication cost is kept minimum at each step. This greedy approach ensures that the solution is optimal with least cost involved and the output is a fully parenthesized product of matrices.