Asymptotic Notation Examples

Common Asymptotic Notations

Following is a list of some common asymptotic notations –

constant	–	$O(1)$
logarithmic	–	$O(\log n)$
linear	–	$O(n)$
n log n	–	$O(n \log n)$
quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$
polynomial	–	$n^{O(1)}$
exponential	–	$2^{O(n)}$

Problem #1: Max Subarray Sub
Given an integer array nums, find the subarray with the largest sum, and return its sum.

Example 1:
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.

Example 2:
Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.

Example 3:
Input: nums = [5,4,-1,7,8]
Output: 23
Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.

Constraints:

1 <= nums.length <= 105
-104 <= nums[i] <= 104

```
class Solution {
    public int maxSubArray(int[] nums) {
        int curSum = nums[0];
        int maxSum = nums[0];
        for(int i = 1; i < nums.length; i++) {
            if(curSum < 0) {
                curSum = 0;
            }
            curSum += nums[i];
            maxSum = Math.max(maxSum, curSum);
        }
        return maxSum;
    }
}
```

Or

1. **Divide the given array in two halves**
2. **Return the maximum of following three**
    a. **Maximum subarray sum in left half (Make a recursive call)**
    b. **Maximum subarray sum in right half (Make a recursive call)**
    c. **Maximum subarray sum such that the subarray crosses the midpoint**

```
def maxCrossingSum(arr, l, m, h):
    sm = 0
    left_sum = -10000
    for i in range(m, l-1, -1):
        sm = sm + arr[i]
        if (sm > left_sum):
            left_sum = sm
    # Include elements on right of mid
    sm = 0
    right_sum = -1000
    for i in range(m, h + 1):
        sm = sm + arr[i]
        if (sm > right_sum):
            right_sum = sm
    # Return sum of elements on left and right of mid
    # returning only left_sum + right_sum will fail for [-2, 1]
    return max(left_sum + right_sum - arr[m], left_sum, right_sum)
```

```python
# Returns sum of maximum sum subarray in aa[l..h]
def maxSubArraySum(arr, l, h):
    #Invalid Range: low is greater than high
    if (l > h):
        return -10000
    # Base Case: Only one element
    if (l == h):
        return arr[l]
    # Find middle point
    m = (l + h) // 2
    # Return maximum of following three possible cases
    # a) Maximum subarray sum in left half
    # b) Maximum subarray sum in right half
    # c) Maximum subarray sum such that the
    #    subarray crosses the midpoint
    return max(maxSubArraySum(arr, l, m-1),
               maxSubArraySum(arr, m+1, h),
               maxCrossingSum(arr, l, m, h))
```

```python
# Driver Code
arr = [2, 3, 4, 5, 7]
n = len(arr)

max_sum = maxSubArraySum(arr, 0, n-1)
print("Maximum contiguous sum is ", max_sum)
```

maxSubArraySum => $T(n) = 2T(n/2) + \Theta(n)$
Time Complexity : O(nlogn)

Problem #2: Strassen Mx Multiplication

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)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underset{X}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \mathrm{X} \underset{Y}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}$$

X , Y and **C** are square metrices ot size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

$$P_1 = A_{11} \cdot (B_{12} - B_{22})$$
$$P_2 = (A_{11} + A_{12}) \cdot B_{22}$$
$$P_3 = (A_{21} + A_{22}) \cdot B_{11}$$
$$P_4 = A_{22} \cdot (B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$
$$P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$
$$P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$$

$$P_5 + P_4 - P_2 + P_6 = C_{11}$$
$$P_1 + P_2 = C_{12}$$
$$P_3 + P_4 = C_{21}$$
$$P_5 + P_1 - P_3 - P_7 = C_{22}$$

$$Z = \begin{bmatrix} I & J \\ K & L \end{bmatrix} \qquad X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Using Strassen's Algorithm compute the following −

$$M_1 := (A + C) \times (E + F)$$

$$M_2 := (B + D) \times (G + H)$$

$$M_3 := (A - D) \times (E + H)$$

$$M_4 := A \times (F - H)$$

$$M_5 := (C + D) \times (E)$$

$$M_6 := (A + B) \times (H)$$

$$M_7 := D \times (G - E)$$

Then,

$$I := M_2 + M_3 - M_6 - M_7$$

$$J := M_4 + M_6$$

$$K := M_5 + M_7$$

$$L := M_1 - M_3 - M_4 - M_5$$

Analysis

$$T(n) = \begin{cases} c & if\, n = 1 \\ 7\, x\, T(\frac{n}{2}) + d\, x\, n^2 & otherwise \end{cases} \quad where\, c\, and\, d\, are\, constants$$

Using this recurrence relation, we get $T(n) = O(n^{log7})$

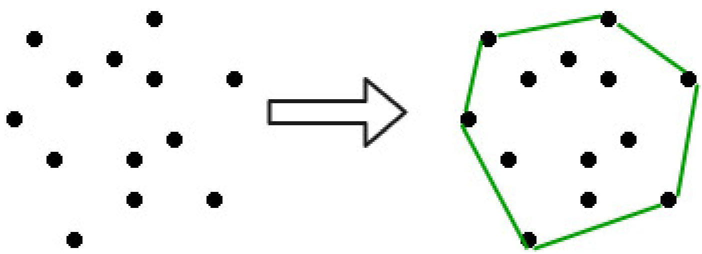
```cpp
void multiply(int a[5][5], int b[5][5], int row, int col, int c1)
{
    int c[5][5];
    //input 0 for all values of c, in order to remove
    //the garbage values assigned earlier
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++)
            c[i][j] = 0;
    }
    //we apply the same formula as above
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            for (int k = 0; k < c1; k++) //columns of first matrix || rows of second matrix
                c[i][j] += a[i][k] * b[k][j];
        }
    }
    //to display matrix
    cout << "\n Matrix c after matrix multiplication is:\n";
    display(c, row, col);
}
```

Divide: The Strassen algorithm takes two square matrices, A and B, and divides them into four equal-sized submatrices. Each matrix is divided into four equal-sized submatrices: A11, A12, A21, A22, B11, B12, B21, B22.

Conquer: The algorithm recursively calculates seven different products using these submatrices, as defined by Strassen's matrix multiplication formula. These products are P1, P2, P3, P4, P5, P6, and P7.

Merge: After obtaining these products, the algorithm constructs the resulting matrix C by using these products and addition/subtraction operations as defined by Strassen's formula.

Problem #3: Convex Hull



Input : points[] = {(0, 0), (0, 4), (-4, 0), (5, 0),

(0, -6), (1, 0)};
Output : (-4, 0), (5, 0), (0, -6), (0, 4)

Divide: We divide the set of n
points into two parts by a vertical line into the left and right halves.
Conquer: We recursively find the convex hull on left and right halves.
Combine or Merge: We combine the left and right convex hull into one convex hull.

Algorithm

1. Before calling the method to compute the convex hull, once and for all, we sort the points by x-coordinate. This step takes O(nlogn) time.
2. Divide Step: Find the point with median x-coordinate. Since the x-coordinate already sorts the input points, this step should take constant time. Depending upon your implementation, sometimes it may take up to O(n) time.
3. Conquer Step: Call the procedure recursively on both halves.
4. Merge Step: Merge the two convex hulls computed by two recursive calls in the conquer step. The merge procedure **should** take O(n) time.

Codes:

```
def convex_hull(points):
    if len(points) == 1:
        return points

    left_half = convex_hull(points[0: len(points)/2])
    right_half = convex_hull(points[len(points)/2:])
    return merge(left_half, right_half)
```

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

```
//strassen algorithm in full(C++):

#include <iostream>
#include <vector>

// Function to add two matrices
std::vector<std::vector<int>> matrixAddition(const std::vector<std::vector<int>>& A, const
std::vector<std::vector<int>>& B) {
    int n = A.size();
    std::vector<std::vector<int>> result(n, std::vector<int>(n));
```

```
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            result[i][j] = A[i][j] + B[i][j];
        }
    }

    return result;
}

// Function to subtract two matrices
std::vector<std::vector<int>> matrixSubtraction(const std::vector<std::vector<int>>& A, const
std::vector<std::vector<int>>& B) {
    int n = A.size();
    std::vector<std::vector<int>> result(n, std::vector<int>(n));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            result[i][j] = A[i][j] - B[i][j];
        }
    }

    return result;
}

// Strassen's matrix multiplication function
std::vector<std::vector<int>> strassenMultiplication(const std::vector<std::vector<int>>& A,
const std::vector<std::vector<int>>& B) {
    int n = A.size();

    // Base case: If the matrix size is 1x1, perform a simple multiplication
    if (n == 1) {
        std::vector<std::vector<int>> C(1, std::vector<int>(1, 0));
        C[0][0] = A[0][0] * B[0][0];
        return C;
    }

    // Divide the matrices into four equal-sized submatrices
    int mid = n / 2;
    std::vector<std::vector<int>> A11(mid, std::vector<int>(mid));
    std::vector<std::vector<int>> A12(mid, std::vector<int>(mid));
    std::vector<std::vector<int>> A21(mid, std::vector<int>(mid));
    std::vector<std::vector<int>> A22(mid, std::vector<int>(mid));
    std::vector<std::vector<int>> B11(mid, std::vector<int>(mid));
    std::vector<std::vector<int>> B12(mid, std::vector<int>(mid));
    std::vector<std::vector<int>> B21(mid, std::vector<int>(mid));
    std::vector<std::vector<int>> B22(mid, std::vector<int>(mid));

    for (int i = 0; i < mid; ++i) {
        for (int j = 0; j < mid; ++j) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + mid];
            A21[i][j] = A[i + mid][j];
            A22[i][j] = A[i + mid][j + mid];
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + mid];
            B21[i][j] = B[i + mid][j];
            B22[i][j] = B[i + mid][j + mid];
        }
    }

    // Recursive calls for submatrix multiplications
    std::vector<std::vector<int>> P1 = strassenMultiplication(A11, matrixSubtraction(B12,
B22));
    std::vector<std::vector<int>> P2 = strassenMultiplication(matrixAddition(A11, A12), B22);
    std::vector<std::vector<int>> P3 = strassenMultiplication(matrixAddition(A21, A22), B11);
    std::vector<std::vector<int>> P4 = strassenMultiplication(A22, matrixSubtraction(B21,
B11));
    std::vector<std::vector<int>> P5 = strassenMultiplication(matrixAddition(A11, A22),
matrixAddition(B11, B22));
```

```
    std::vector<std::vector<int>> P6 = strassenMultiplication(matrixSubtraction(A12, A22),
matrixAddition(B21, B22));
    std::vector<std::vector<int>> P7 = strassenMultiplication(matrixSubtraction(A11, A21),
matrixAddition(B11, B12));

    // Compute the submatrices of the result
    std::vector<std::vector<int>> C11 = matrixSubtraction(matrixAddition(matrixAddition(P5,
P4), P6), P2);
    std::vector<std::vector<int>> C12 = matrixAddition(P1, P2);
    std::vector<std::vector<int>> C21 = matrixAddition(P3, P4);
    std::vector<std::vector<int>> C22 = matrixSubtraction(matrixSubtraction(matrixAddition(P5,
P1), P3), P7);

    // Combine the submatrices to form the result
    std::vector<std::vector<int>> C(n, std::vector<int>(n, 0));
    for (int i = 0; i < mid; ++i) {
        for (int j = 0; j < mid; ++j) {
            C[i][j] = C11[i][j];
            C[i][j + mid] = C12[i][j];
            C[i + mid][j] = C21[i][j];
            C[i + mid][j + mid] = C22[i][j];
        }
    }

    return C;
}

int main() {
    std::vector<std::vector<int>> A = {{1, 2}, {3, 4}};
    std::vector<std::vector<int>> B = {{5, 6}, {7, 8}};

    std::vector<std::vector<int>> result = strassenMultiplication(A, B);

    int n = result.size();
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            std::cout << result[i][j] << " ";
        }
        std::cout << "\n";
    }

    return 0;
}
```