Divide & Conquer

> *Solve the problem instance by divide it into smaller instances of the same type*

Divide → Conquer → Combine

- Divide: Split the problem into smaller pieces of the same type.
- Conquer: If the problem is small enough, solve it.
- Combine: Combine the result of each smaller problem into a result of original problem.

Binary Search

> *A way to search the item in (sorted) list efficiently.*

```
bool binary_search_vector(int l, int r, int x, vector<int> &v) {
    if (l == r) return v[l] == x;
    int m = (l + r) >> 1;
    return (v[m]<=x)
            ? binary_search_vector(l, m, x, v)
            : binary_search_vector(m+1, r, x, v);
}
```

Time Complexity: $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(1)$

Note: The array needs to be sorted.

Merge Sort

> *A way to sort with $\Theta(n\ log\ n)$ complexity is by partitioning the array into 2 parts of (almost) same size*

- Divide: Split the array into 2 parts (1) start to middle (2) middle + 1 to end.
- Conquer: If the array is small enough $(n = 1)$, it is already sorted.
- Combine: Combine 2 sorted arrays in $\Theta(n)$

Combine Sorted Array Part

```
void combine_sorted_arry(int l, int r, vector<int> &v) {
    int m = (l+r) >> 1;

    vector<int> v1;
    vector<int> v2;
    v1.assign(v.begin()+l, v.begin()+m+1);
    v2.assign(v.begin()+m+1, v.begin()+r+1);
```

```
    int i = 0, j = 0, index = l;
    while (i<v1.size() || j<v2.size()) {
        if (i<v1.size() && j<v2.size()) {
            if (v1[i]<v2[j]) {
                v[index] = v1[i];
                index++, i++;
            } else {
                v[index] = v2[j];
                index++, j++;
            }
        } else {
            if (i<v1.size()) {
                v[index] = v1[i];
                index++, i++;
            } else {
                v[index] = v2[j];
                index++, j++;
            }
        }
    }
}
```

Merge Sort Part

```
void merge_sort(int l, int r, vector<int> &v) {
    if (l == r) return ;
    int m = (l+r) >> 1;
    merge_sort(l, m, v);
    merge_sort(m+1, r, v);
    combine_sorted_arry(l, r, v);
}
```

Time Complexity: $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$

Quick Sort

> *A sorting algorithm using pivot to divide array into 2 parts (subproblems maybe different size).*

- Divide
  - Subproblem 1 must contains only $A[i] <= \text{pivot}$
  - Subproblem 2 must contains only $A[i] >= \text{pivot}$
  - Pivot can be either on subproblem 1 and subproblem 2
- Conquer
  - Because every data in subproblem 1 is less than elements of subproblem 2, we can conquer by just append subproblem 2 to subproblem 1

Hoare's Algorithm

```
int hoare_partition(int l, int r, vector<int> &v) {
    int pivot = v[l + rand() % (r - l)];
```

```
    int start = l - 1;
    int stop = r + 1;

    while (true) {
        do { start += 1; } while (v[start] < pivot);
        do { stop -= 1; } while (v[stop] > pivot);
        if (start >= stop) return stop;
        swap(v[start], v[stop]);
    }
}
```

Quick Sort

```
void quick_sort(int l, int r, vector<int> &v) {
    if (l == r) return ;
    int pivot = hoare_partition(l, r, v);
    quick_sort(l, pivot, v);
    quick_sort(pivot+1, r, v);
}
```

Time Complexity:

- Average Case: $O(n\ log\ n)$
- Worst Case: $O(n^2)$

Modulo Exponential

> *A way to compute x^n in logarithmic time by dividing the n into 2 parts of (almost) same size.*

$$f(x,n) = \begin{cases} 1 & ; n = 0 \\ x & ; n = 1 \\ x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} & ; n\ is\ even \\ x^{\lfloor \frac{n}{2} \rfloor} \cdot x^{\lfloor \frac{n}{2} \rfloor} \cdot x & ; n\ is\ odd \end{cases}$$

```
int modular_exponent(int x, int n, int m = 1e9 + 7) {
    if (n == 0) return 1;
    if (n == 1) return x % m;

    int ans = modular_exponent(x, n/2) % m;

    return (n%2) ? (((ans*ans)%m)*x)%m : (ans*ans)%m;
}
```

Time Complexity: $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(1)$

Maximum Subarray Sum

> *The maximum sum of a continuous subarray of A is the problem of choosing positions l and r that maximize $\sum_{i=l}^{r} A[i]$.*

```
int query(int l, int r) {
    return qs[r] - qs[l-1];
}

int maximum_subarray_sum(int l, int r, int arr[]) {
    if (l == r) return arr[l];
    int m = (l + r) >> 1;

    int r1 = maximum_subarray_sum(l, m, arr);
    int r2 = maximum_subarray_sum(m+1, r, arr);

    int mx1 = INT_MIN;
    int mx2 = INT_MIN;
    for (int i=l; i<=m; i++) mx1 = max(mx1, query(i, m));
    for (int i=m+1; i<=r; i++) mx2 = max(mx2, query(m+1, i));

    return max(mx1+mx2, max(r1, r2));
}
```

Time Complexity: $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$

Note: We can improve it better by using [Kadane's Algorithm]

Strassen's Matrix Multiplication

> *A way to multiply matrix which has $2^k \times 2^k$ dimensions with $O(n^{2.807})$ is invented by Volker Strassen.*

Let A, B be the matrix of $2^k \times 2^k$ dimensions, and $C = AB$

- $A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$
- $B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$

Strassen's Algorithm

Before we calculate AB we need to compute matrix $M_1 \dots M_7$

- $M_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$
- $M_2 = (A_{2,1} + A_{2,2})(B_{1,1})$
- $M_3 = (A_{1,1})(B_{1,2} - B_{2,2})$
- $M_4 = (A_{2,2})(B_{2,1} - B_{1,1})$
- $M_5 = (A_{1,1} + A_{1,2})(B_{2,2})$
- $M_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$
- $M_7 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$

To calculate C matrix

- $C_{1,1} = M_1 + M_4 - M_5 + M_7$

- $C_{1,2} = M_3 + M_5$
- $C_{2,1} = M_2 + M_4$
- $C_{2,2} = M_1 + M_2 + M_3 + M_6$

  ℹ️ Total of 7 matrix multiplication and 18 matrix addition.

```
Matrix strassen(Matrix A, Matrix B) {
    if (A.size() == 1) {
        return {{A[0][0] * B[0][0]}};
    }

    int n = A.size() >> 1;
    Matrix A11(n, vector<int>(n)), A12(n, vector<int>(n)), A21(n, vector<int>(n)),
A22(n, vector<int>(n));
    Matrix B11(n, vector<int>(n)), B12(n, vector<int>(n)), B21(n, vector<int>(n)),
B22(n, vector<int>(n));
    Matrix C(n << 1, vector<int>(n << 1));

    split_matrix(A, A11, A12, A21, A22);
    split_matrix(B, B11, B12, B21, B22);

    Matrix M1 = strassen(add_matrix(A11, A22), add_matrix(B11, B22));
    Matrix M2 = strassen(add_matrix(A21, A22), B11);
    Matrix M3 = strassen(A11, sub_matrix(B12, B22));
    Matrix M4 = strassen(A22, sub_matrix(B21, B11));
    Matrix M5 = strassen(add_matrix(A11, A12), B22);
    Matrix M6 = strassen(sub_matrix(A21, A11), add_matrix(B11, B12));
    Matrix M7 = strassen(sub_matrix(A12, A22), add_matrix(B21, B22));

    Matrix C11 = add_matrix(sub_matrix(add_matrix(M1, M4), M5), M7);
    Matrix C12 = add_matrix(M3, M5);
    Matrix C21 = add_matrix(M2, M4);
    Matrix C22 = add_matrix(sub_matrix(add_matrix(M1, M3), M2), M6);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = C11[i][j];
            C[i][j+n] = C12[i][j];
            C[i+n][j] = C21[i][j];
            C[i+n][j+n] = C22[i][j];
        }
    }

    return C;
}
```

Time Complexity: $O(n^{log_2 7})$

Note: This algorithm is galactic algorithm which need to use with large N.

Closest Pair

> *To find the minimum distance between 2 points in the plane.*

- Divide: Split the data into 2 parts
- Combine: Calculate distance of the point across 2 parts.
- Conquer: If the number of points is one return with `inf` value.

Note: Array of points need to be sorted.

```
long long get_distance(pair<int, int> a, pair<int, int> b) {
    long long dx = (a.first - b.first);
    long long dy = (a.second - b.second);
    return dx*dx + dy*dy;
}
```

```
long long find_closest_distance(int l, int r) {
    if (l == r) return 1e18;
    int m = (l + r) >> 1;
    long long ans = min(find_closest_distance(l, m), find_closest_distance(m+1,
r));

    for (int i = max(l, m-8); i <= m; i++) {
        for (int j = m + 1; j <= min(r, m+8); j++) {
            ans = min(ans, get_distance(points[i], points[j]));
        }
    }

    return ans;
}
```

Time Complexity: $\Theta(n \ log \ n)$

ℹ This problem can be solve by using Sweep line technique.

Celebrity Problem

> *To find celebrity - a person who does not know anyone, but is known by everyone - in the party of N persons.*

Note: knowing relation for a pair of people is not symmetric, e.i. it is possible that A knows B but B does not know A.

Let B is a knowing matrix which $B[i][j]$ represents "Does i know j ?"

- Output: A number X such that $B[*][X] = \text{true}$ and $B[X][*] = \text{false}$
- -1 if no such X exist.

Observation

We can use these observations to form the recurrent function.

- If `B[i][*] = true` then person `i` is not celeb.
- If `B[*][i] = false` then person `i` is not celeb.

```
int find_celeb(int start, int stop) {
    if (start == stop) {
        for (int i = 1; i <= n; i++) {
            if (B[start][i]) return -1;
            if (i != start && !B[i][start]) return -1;
        }
        return start;
    }

    if (B[start][stop]) {
        return find_celeb(start+1, stop);
    } else {
        return find_celeb(start, stop+1);
    }
}
```

Time Complexity: $O(n)$