INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

INGENIERÍA EN SISTEMAS COMPUTACIONALES

UNIDAD DE APRENDIZAJE: ANALISIS Y DISEÑO DE ALGORITMOS

DOCENTE: GARCIA FLORIANO ANDRES

EJERCICO PRACTICO 4: DaC(Divide and Conquer)

ALUMNOS:

López Martínez Jonathan

Mejía Mejía Mauricio Xavier

Nava Montiel Erasmo

Silva Vázquez Diego

GRUPO: 3CV5

FECHA DE ENTREGA: 29/Octubre /2025

1.Quick Select (Encontrar el k-ésimo elemento más pequeño de un arreglo).

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // Necesario para time(), srand() y clock_gettime

/* --- Funciones de Medición y RNG (de tu otro código) --- */

// Mide el tiempo actual en milisegundos usando CLOCK_MONOTONIC
static double now_ms() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (double)ts.tv_sec * 1000.0 + (double)ts.tv_nsec / 1e6;
}

// RNG simple para enteros
static inline int fast_rand() {
    return (rand() ^ (rand() << 15)) & 0x7fffffff; // no negativo
}

/* --- Funciones del Algoritmo Quickselect (sin cambios) --- */

// Funcion para intercambiar dos elementos
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Funcion de particion (usa el pivote al final)
int partition(int a[], int l, int r) {
    int pivot = a[r];
    int i = l;
    for (int j = l; j < r; j++) {
        if (a[j] < pivot) {
            swap(&a[i], &a[j]);
            i++;
        }
    }
```

```c
    swap(&a[i], &a[r]);
    return i; // posicion final del pivote
}

// Selecciona un pivote aleatorio para reducir el peor caso
int randomized_partition(int a[], int l, int r) {
    // Evita división por cero si l == r
    int range = r - l + 1;
    if (range <= 0) range = 1;

    int p = l + rand() % range;
    swap(&a[p], &a[r]);
    return partition(a, l, r);
}

// Algoritmo Quickselect: encuentra el k-esimo menor elemento (k es 0-index)
int quickselect(int a[], int l, int r, int k) {
    while (l <= r) {
        // Caso base: si solo queda un elemento
        if (l == r) {
            return a[l];
        }

        int p = randomized_partition(a, l, r);

        if (p == k) {
            return a[p]; // el pivote esta en la posicion correcta
        } else if (p > k) {
            r = p - 1; // buscar en la parte izquierda
        } else {
            l = p + 1; // buscar en la parte derecha
        }
    }
    return -1; // No debería llegar aquí si k es válido
}
int main() {
    // Tamaños de prueba requeridos
    const int sizes[] = {100, 1000, 100000};
    int num_tests = sizeof(sizes) / sizeof(sizes[0]);

    // Inicializar semilla para TODAS las funciones rand()
    srand((unsigned)time(NULL));

    printf("Ejecutando Pruebas para Quick Select...\n");

    for (int s = 0; s < num_tests; s++) {
        int n = sizes[s];

        int *a = (int*)malloc(sizeof(int) * n);
        if (a == NULL) {
            fprintf(stderr, "Error al alocar memoria para n=%d\n", n);
            return 1;
        }
```

```c
        // Llenar el arreglo con números aleatorios
        for (int i = 0; i < n; i++) {
            a[i] = fast_rand();
        }

        // --- Prueba: Encontrar la mediana ---
        // 'k' es 0-index, así que n/2 es un buen objetivo
        int k_idx = n / 2;

        double start = now_ms();

        // Llamada al algoritmo
        int resultado = quickselect(a, 0, n - 1, k_idx);

        double elapsed = now_ms() - start;

        // Imprimir en el formato deseado
        printf("n=%6d -> k-esimo(%d)=%d, tiempo=%.2f ms\n", n, k_idx, resultado, elapsed);

        free(a);
    }

    return 0;
}
```

Resultados:

```
PS C:\Users\tptme\Downloads> cd "c:\Users\tptme\Downloads\" ; if ($?) { gcc k_esimo.c
Ejecutando Pruebas para Quick Select...
n=   100 -> k-esimo(50)=481492673, tiempo=0.00 ms
n=  1000 -> k-esimo(500)=552595861, tiempo=0.01 ms
n=100000 -> k-esimo(50000)=538253773, tiempo=1.44 ms
```

Este programa, encuentra el k-ésimo elemento más pequeño en un arreglo.

Hace lo siguiente:

1. Genera automáticamente arreglos de tamaños 100, 1000 y 100000 con números aleatorios.
2. Mide el tiempo de ejecución en milisegundos (ms) usando una función de alta precisión (now_ms).
3. Prueba el algoritmo buscando un elemento específico (en este caso, la mediana, k = n / 2).
4. Imprime los resultados en un formato limpio,

Estrategia "Divide y Vencerás" (DaC)

La estrategia del algoritmo sigue siendo la misma:

1. Dividir: Se elige un pivote aleatorio (randomized_partition).
2. Vencer: Se particiona el arreglo. El pivote queda en su "posición final ordenada", p.
3. Seleccionar (en lugar de Combinar): Se compara p con el índice k que buscamos.
    - Si p == k, se encontró el elemento.
    - Si p > k, se descarta la parte derecha y se busca solo a la izquierda.
    - Si p < k, se descarta la parte izquierda y se busca solo a la derecha.

Fragmentos Clave del Código (C)

Las funciones swap, partition, randomized_partition y quickselect son idénticas a las de la versión anterior.

1. Las Funciones de Utilidad (Añadidas)

C

```
static double now_ms() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts); // <-- Reloj de alta precisión
    return (double)ts.tv_sec * 1000.0 + (double)ts.tv_nsec / 1e6;
}

static inline int fast_rand() {
    return (rand() ^ (rand() << 15)) & 0x7fffffff; // <-- Generador simple
}
```

- Punto Clave: now_ms es la función más importante para tu reporte. Permite medir el tiempo de pared real en milisegundos, lo cual es exactamente lo que necesitas para comparar el rendimiento.

2. La Función quickselect (El Núcleo de DaC)

C

```
int quickselect(int a[], int l, int r, int k) {
    while (l <= r) {
        // ... (manejo del caso base l == r) ...

        // 1. DIVIDIR Y VENCER
        int p = randomized_partition(a, l, r);
```

```
        // 2. SELECCIONAR
        if (p == k) {
            return a[p]; // Encontrado
        } else if (p > k) {
            r = p - 1; // Buscar a la izquierda
        } else {
            l = p + 1; // Buscar a la derecha
        }
    }
    return -1; // Error
}
```

- Punto Clave: La lógica de "descartar" la mitad del arreglo en cada iteración (r = p - 1 o l = p + 1) es lo que le da al algoritmo su complejidad promedio de $O(n)$.

3. El Nuevo main (Centro de Pruebas)

Esta función ahora maneja todo el experimento.

C

```
int main() {
    // Tamaños de prueba requeridos
    const int sizes[] = {100, 1000, 100000};
    int num_tests = sizeof(sizes) / sizeof(sizes[0]);

    srand((unsigned)time(NULL)); // Semilla para rand()

    for (int s = 0; s < num_tests; s++) {
        int n = sizes[s];

        int *a = (int*)malloc(sizeof(int) * n);
        // ... (Llenar 'a' con fast_rand()) ...

        // --- Prueba: Encontrar la mediana ---
        int k_idx = n / 2; // (k es 0-index)

        double start = now_ms();
        int resultado = quickselect(a, 0, n - 1, k_idx);
        double elapsed = now_ms() - start;

        // Imprimir en el formato deseado
        printf("n=%6d -> k-esimo(%d)=%d, tiempo=T%.2f ms\n", n, k_idx, resultado, elapsed);

        free(a);
```

```
    }
    return 0;
}
```

- Puntos Clave:
    - const int sizes[]: Define los casos de prueba que te pidieron (100, 1000, 100000).
    - k_idx = n / 2;: Elegiste probar buscando la mediana. Esta es una buena prueba, ya que obliga al algoritmo a ejecutarse hasta encontrar un elemento central.
    - start = now_ms() / elapsed = now_ms() - start: Es el par de líneas clave para la medición de tiempo.

2. Contar inversiones (en un arreglo, determina el número de pares i<j tales que arr[i]>arr[j].

```c
// inversions_dac.c
// Compile:  gcc -O2 -std=c11 inversions_dac.c -o inversions -lrt (puede necesitar -lrt por clock_gettime)
// Run:      ./inversions
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdint.h>

/**
 * @brief Función recursiva de DaC que ordena y cuenta inversiones.
 * * @param a     Arreglo de trabajo (se modifica en el lugar).
 * @param tmp   Arreglo temporal (buffer) para la fusión.
 * @param l     Índice izquierdo (inclusivo).
 * @param r     Índice derecho (exclusivo).
 * @return      Número de inversiones (long long).
 */
static long long merge_count(int *a, int *tmp, int l, int r) {
    // Caso base: 0 o 1 elemento, no hay inversiones.
    if (r - l <= 1) return 0;

    // Dividir
    int m = (l + r) / 2;

    // Vencer (Recursión en las mitades)
    long long inv = merge_count(a, tmp, l, m) + merge_count(a, tmp, m, r);

    // Combinar (Fusión y conteo de inversiones de cruce)
    int i = l, j = m, k = l;
    while (i < m && j < r) {
        if (a[i] <= a[j]) {
            tmp[k++] = a[i++];
        } else {
            // Si a[j] < a[i], es una inversión.
            // Y como la mitad izquierda (i...m) está ordenada,
            // todos los elementos restantes de la izquierda (m-i)
            // también son mayores que a[j].
                tmp[k++] = a[j++];
                inv += (long long)(m - i); // <-- El conteo clave
            }
        }

        // Copiar elementos restantes de la izquierda (si los hay)
        while (i < m) tmp[k++] = a[i++];
        // Copiar elementos restantes de la derecha (si los hay)
        while (j < r) tmp[k++] = a[j++];

        // Copiar el resultado de tmp de vuelta al arreglo 'a'
        for (i = l; i < r; ++i) a[i] = tmp[i];

        return inv;
    }
```

```c
/**
 * @brief Función "wrapper" para contar inversiones.
 * * Aloca la memoria temporal e inicia la recursión.
 */
static long long count_inversions(int *a, int n) {
    // --- CORRECCIÓN AQUÍ ---
    // 'tmp' debe ser un puntero (int*)
    int *tmp = (int*)malloc((size_t)n * sizeof(int));
    if (!tmp) {
        fprintf(stderr, "No se pudo reservar memoria para tmp\n");
        exit(1);
    }

    long long inv = merge_count(a, tmp, 0, n);

    free(tmp); // Liberar la memoria temporal
    return inv;
}

// RNG simple para enteros grandes (combina dos rand()).
static inline int fast_rand() {
    return (rand() ^ (rand() << 15)) & 0x7fffffff; // no negativo
}

// Mide el tiempo actual en milisegundos usando CLOCK_MONOTONIC
static double now_ms() {
    struct timespec ts;
    // CLOCK_MONOTONIC es un reloj que no se ve afectado por cambios
    // en la hora del sistema, ideal para medir intervalos.
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (double)ts.tv_sec * 1000.0 + (double)ts.tv_nsec / 1e6;
}
int main(void) {
    const int sizes[] = {100, 1000, 100000};
    srand(123); // semilla fija para reproducibilidad

    for (int s = 0; s < 3; ++s) {
        int n = sizes[s];

        
        int *a = (int*)malloc((size_t)n * sizeof(int));
        if (!a) {
            fprintf(stderr, "Sin memoria para a\n");
            return 1;
        }

        // Llenar el arreglo con números aleatorios
        for (int i = 0; i < n; ++i) a[i] = fast_rand();

        double start = now_ms();
        long long inv = count_inversions(a, n); // Llamada al algoritmo
        double elapsed = now_ms() - start;

        printf("n=%6d -> inversiones=%lld, tiempo=%.2f ms\n", n, inv, elapsed);

        free(a); // Liberar la memoria del arreglo principal
    }
    return 0;
}
```

Resultados:

```
PS C:\Users\tptme\Downloads> cd "c:\Users\tptme\Downloads\" ; if ($?) { gcc contar_inversiones.c
n=   100 -> inversiones=2487, tiempo=0.01 ms
n=  1000 -> inversiones=250227, tiempo=0.09 ms
n=100000 -> inversiones=2504440118, tiempo=14.18 ms
```

Este programa cuenta el número de "inversiones" en un arreglo. Una inversión es un par de índices (i, j) tales que i < j (el índice i viene antes que j) pero el valor a[i] > a[j] (el valor en i es mayor que el valor en j).

Es una medida de cuán "desordenado" está un arreglo. Un arreglo perfectamente ordenado tiene 0 inversiones.

- Ejemplo: En el arreglo [3, 1, 2], las inversiones son:
    - (3, 1) (índices 0, 1)
    - (3, 2) (índices 0, 2)
- Total de inversiones: 2.

Estrategia "Divide y Vencerás" (DaC)

Este algoritmo es una modificación muy inteligente del algoritmo de ordenamiento Merge Sort.

1. Dividir: Divide el arreglo en dos mitades exactas (izquierda y derecha).
2. Vencer (Conquer): Llama recursivamente a la función para:
    - inv_izq: Contar todas las inversiones *dentro* de la mitad izquierda.
    - inv_der: Contar todas las inversiones *dentro* de la mitad derecha.
3. Combinar: Este es el paso crucial. Mientras se fusionan (merge) las dos mitades (que ahora ya están ordenadas), se cuentan las "inversiones de cruce" (inv_cruce). Estas son las inversiones (i, j) donde i está en la mitad izquierda y j en la derecha.
4. Resultado: El total de inversiones es inv_izq + inv_der + inv_cruce.

Fragmentos Clave del Código (C)

Tu implementación en C utiliza la función merge_count como el núcleo de DaC, y count_inversions como una función "wrapper" o envoltorio.

1. La Función "Wrapper": count_inversions

Esta función prepara el terreno para la recursión.

C

```
static long long count_inversions(int *a, int n) {
    // CORRECCIÓN: Se aloca un puntero (int*)
    int *tmp = (int*)malloc((size_t)n * sizeof(int));
    if (!tmp) {
        fprintf(stderr, "No se pudo reservar memoria para tmp\n");
        exit(1);
    }

    // Inicia la recursión
    long long inv = merge_count(a, tmp, 0, n);

    free(tmp); // Libera la memoria
    return inv;
}
```

- Punto Clave: Su trabajo principal es alocar el arreglo temporal tmp una sola vez. Esto es mucho más eficiente que alocar memoria nueva en cada llamada recursiva. Luego, libera la memoria al final.

2. El Núcleo de DaC (Dividir y Vencer): merge_count

Esta es la función recursiva que hace todo el trabajo.

C

```
static long long merge_count(int *a, int *tmp, int l, int r) {
    // Caso base: 0 o 1 elemento, no hay inversiones.
    if (r - l <= 1) return 0;

    // Dividir
    int m = (l + r) / 2;

    // Vencer (Recursión en las mitades)
    long long inv = merge_count(a, tmp, l, m) + merge_count(a, tmp, m, r);
    ...
```

- Punto Clave: Las dos llamadas recursivas (merge_count(...) + merge_count(...)) resuelven los subproblemas. La variable inv acumula las inversiones encontradas *dentro* de la mitad izquierda y *dentro* de la mitad derecha.

3. El Núcleo de DaC (Combinar y Contar): merge_count

Esta es la continuación de la función anterior y es la parte más importante del algoritmo.

C

```
    ...
    // Combinar (Fusión y conteo de inversiones de cruce)
    int i = l, j = m, k = l;
    while (i < m && j < r) {
        if (a[i] <= a[j]) {
            tmp[k++] = a[i++];
        } else {
            // Si a[j] < a[i], es una inversión.
            tmp[k++] = a[j++];

            // --- LA LÍNEA CLAVE ---
            inv += (long long)(m - i);
        }
    }

    // Copiar elementos restantes (si los hay)
    while (i < m) tmp[k++] = a[i++];
    while (j < r) tmp[k++] = a[j++];

    // Copiar el resultado de tmp de vuelta al arreglo 'a'
    for (i = l; i < r; ++i) a[i] = tmp[i];

    return inv;
}
```

- Punto Clave (La Línea Mágica): inv += (long long)(m - i);
    - ¿Por qué funciona? Esta línea se ejecuta cuando encontramos un elemento en la mitad derecha (a[j]) que es *menor* que un elemento en la mitad izquierda (a[i]).
    - Debido a que las llamadas recursivas *ya ordenaron* la mitad izquierda (desde l hasta m-1), sabemos que si a[j] es menor que a[i], también debe ser menor que todos los elementos restantes en la mitad izquierda (es decir, a[i+1], a[i+2], ..., a[m-1]).
    - El número de esos elementos restantes es exactamente (m - i).
    - Por lo tanto, esta sola línea de código cuenta (m - i) inversiones de cruce de un solo golpe.
- Punto Clave (Copiar de vuelta): El for loop al final (for (i = l; i < r; ++i) a[i] = tmp[i];) es esencial. Copia el sub-arreglo ordenado del búfer tmp de vuelta al arreglo original a. Esto asegura que cuando la recursión "suba" un nivel, esté trabajando con mitades ya ordenadas.

4. Medición de Tiempo

C

```
static double now_ms() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (double)ts.tv_sec * 1000.0 + (double)ts.tv_nsec / 1e6;
}
```

- Punto Clave: El uso de clock_gettime(CLOCK_MONOTONIC, ...) es una excelente práctica. Mide el "tiempo de pared" (tiempo real transcurrido) y no se ve afectado si el reloj del sistema cambia. Es mucho más preciso para el *benchmarking* (medición de rendimiento) que clock() de time.h.

3. Par de puntos más cercanos (puntos en 2d con la menor distancia euclidiana).

C: > Users > tptme > Downloads > C practica4_Puntos.c > compareY(const void *, const void *)

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <time.h>

// Estructura para un punto en 2D
struct Point {
    double x, y;
};

/* --- Funciones de Comparaci□n para qsort --- */

// Comparar por coordenada x
int compareX(const void* a, const void* b) {
    struct Point *p1 = (struct Point *)a;
    struct Point *p2 = (struct Point *)b;
    if (p1->x < p2->x) return -1;
    if (p1->x > p2->x) return 1;
    return 0;
}

// Comparar por coordenada y
int compareY(const void* a, const void* b) {
    struct Point *p1 = (struct Point *)a;
    struct Point *p2 = (struct Point *)b;
    if (p1->y < p2->y) return -1;
    if (p1->y > p2->y) return 1;
    return 0;
}

/* --- Funciones de Utilidad --- */

// Calcular distancia euclidiana
double dist(struct Point p1, struct Point p2) {
    return sqrt((p1.x - p2.x) * (p1.x - p2.x) +
                (p1.y - p2.y) * (p1.y - p2.y));
}

// Caso base: Fuerza bruta para n <= 3
double bruteForce(struct Point P[], int n) {
    double min_dist = DBL_MAX;
    int i,j;
    for ( i = 0; i < n; ++i)
        for ( j = i + 1; j < n; ++j)
            if (dist(P[i], P[j]) < min_dist)
                min_dist = dist(P[i], P[j]);
    return min_dist;
}
```

```
// Funci?n para encontrar el m?nimo de dos dobles
double min(double x, double y) {
    return (x < y) ? x : y;
}

/* --- Funciones D&V Principales --- */

// Encuentra la distancia m?nima en la franja (strip)
// La franja ya est? ordenada por y
double stripClosest(struct Point strip[], int size, double d) {
    double min_dist = d;

    // Ordenar la franja por 'y' (Este es el paso O(N log N) en la combinaci?n)
    qsort(strip, size, sizeof(struct Point), compareY);

    int i, j;
    // Iterar y comparar con los vecinos cercanos
    for ( i = 0; i < size; ++i) {
        for ( j = i + 1; j < size && (strip[j].y - strip[i].y) < min_dist; ++j) {
            if (dist(strip[i], strip[j]) < min_dist)
                min_dist = dist(strip[i], strip[j]);
        }
    }
    return min_dist;
}

// Funci?n recursiva principal de Divide y Vencer?s
double closestUtil(struct Point Px[], int n) {
    // Caso base
    if (n <= 3)
        return bruteForce(Px, n);

    // Dividir
    int mid = n / 2;
    struct Point midPoint = Px[mid];

    // Vencer (Recursi?n)
    // Usamos aritm?tica de punteros para pasar las mitades
    double dl = closestUtil(Px, mid);
    double dr = closestUtil(Px + mid, n - mid);

    double d = min(dl, dr);

    // Combinar (Construir la franja)
    // Usamos malloc para evitar desbordamiento de pila (stack overflow) con N grande
    struct Point* strip = (struct Point*)malloc(n * sizeof(struct Point));
    if (strip == NULL) {
        fprintf(stderr, "Error de alocaci?n de memoria\n");
        exit(1);
    }

    int j = 0;
    int i;
    for ( i = 0; i < n; i++)
        if (fabs(Px[i].x - midPoint.x) < d)
            strip[j] = Px[i], j++;

    // Encontrar el m?nimo en la franja y liberar memoria
    double d_strip = stripClosest(strip, j, d);
    free(strip);

    return min(d, d_strip);
}
```

```c
// Funci□n principal (Wrapper)
double closest(struct Point P[], int n) {
    // 1. Pre-ordenamiento por x
    qsort(P, n, sizeof(struct Point), compareX);

    // 2. Llamada recursiva
    return closestUtil(P, n);
}

/* --- Main y Pruebas --- */

int main() {
    int N_values[] = {10, 100, 1000, 100000};
    int num_tests = sizeof(N_values) / sizeof(N_values[0]);

    // Inicializar semilla para n□meros aleatorios
    srand(time(NULL));

    int i,j;
    for ( i = 0; i < num_tests; i++) {
        int N = N_values[i];
        struct Point* P = (struct Point*)malloc(N * sizeof(struct Point));
        if (P == NULL) {
            fprintf(stderr, "Error al alocar P\n");
            return 1;
        }

        // Generar N puntos aleatorios (entre 0 y 10000)
        for ( j = 0; j < N; j++) {
            P[j].x = (double)rand() / (double)(RAND_MAX / 10000.0);
            P[j].y = (double)rand() / (double)(RAND_MAX / 10000.0);
        }

        clock_t start = clock();
        double min_dist = closest(P, N);
        clock_t end = clock();

        double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

        printf("N = %-7d | Dist. Minima: %-10.6f | Tiempo (s): %-10.6f\n", N, min_dist, time_taken);

        free(P);
    }

    return 0;
}

```

RESULTADOS:

```
PS C:\Users\tptme\Downloads> cd "c:\Users\tptme\Downloads\" ; if ($?) { gcc practica4_Puntos.c -
N = 10      | Dist. Minima: 561.698618 | Tiempo (s): 0.000000
N = 100     | Dist. Minima: 81.612908  | Tiempo (s): 0.000000
N = 1000    | Dist. Minima: 10.108033  | Tiempo (s): 0.000000
N = 100000  | Dist. Minima: 0.000000   | Tiempo (s): 0.052000
PS C:\Users\tptme\Downloads>
```

Este programa encuentra el par de puntos, en un conjunto de puntos 2D, que tienen la distancia euclidiana más pequeña entre ellos.

Implementa un algoritmo de Divide y Vencerás (DaC) que es mucho más eficiente (complejidad O(n (\log n)^2) o O(n \log n), dependiendo de la implementación) que la fuerza bruta (que sería O(n^2)).

Estrategia "Divide y Vencerás" (DaC)

El algoritmo funciona siguiendo los pasos clásicos de DaC:

1. (Pre-paso): El algoritmo requiere que todos los puntos estén ordenados por su coordenada x. Esto se hace una sola vez al inicio.
2. Dividir: Se traza una línea vertical imaginaria (midPoint) que divide el conjunto de puntos en dos mitades de igual tamaño (mitad izquierda y mitad derecha).
3. Vencer (Conquer): Se llama recursivamente al mismo algoritmo en la mitad izquierda (para encontrar su distancia mínima, dl) y en la mitad derecha (para encontrar dr).
4. Combinar: Este es el paso clave.
    - Se toma la distancia mínima de las dos mitades: d = min(dl, dr).
    - El problema: El par de puntos más cercano podría estar "cruzando la línea", con un punto en la mitad izquierda y otro en la derecha.
    - Para solucionar esto, se crea una "franja" (strip) que contiene solo los puntos que están a una distancia menor que d de la línea central.
    - Se busca la distancia mínima dentro de esta franja (d_strip) y se compara con d. La respuesta final es min(d, d_strip).

Fragmentos Clave del Código (C)

Aquí están las partes más importantes de tu implementación en C:

1. La Función "Wrapper": closest()

Esta es la función que el main llama. Su único trabajo es realizar el pre-ordenamiento crucial por la coordenada x. Sin este paso, la estrategia de "Dividir" no funcionaría.

C

```
// Funcin principal (Wrapper)
double closest(struct Point P[], int n) {
    // 1. Pre-ordenamiento por x
    qsort(P, n, sizeof(struct Point), compareX);
```

2. Llamada recursiva

return closestUtil(P, n);

}

- Punto Clave: El qsort inicial que usa compareX es el pre-requisito indispensable del algoritmo. Garantiza que la división en mid sea $O(1)$ (solo tomar el índice) y sea balanceada.

2. El Caso Base: bruteForce()

Toda función recursiva de DaC necesita una condición de "parada". Cuando el problema es lo suficientemente pequeño (en este caso, 3 puntos o menos), se resuelve por fuerza bruta, ya que es más rápido que seguir dividiendo.

C

```
// Caso base: Fuerza bruta para n <= 3
double bruteForce(struct Point P[], int n) {
    double min_dist = DBL_MAX;
    int i,j;
    for ( i = 0; i < n; ++i)
        for ( j = i + 1; j < n; ++j)
            if (dist(P[i], P[j]) < min_dist)
                min_dist = dist(P[i], P[j]);
    return min_dist;
}
```

- Punto Clave: Esta función detiene la recursión y su complejidad es $O(n^2)$, pero como n es constante (siempre $n \le 3$), su tiempo de ejecución es $O(1)$.

3. La Función Recursiva Principal: closestUtil()

Este es el corazón de la lógica "Dividir y Vencer".

```
double closestUtil(struct Point Px[], int n) {
    // Caso base
    if (n <= 3)
        return bruteForce(Px, n);

    // Dividir
    int mid = n / 2;
    struct Point midPoint = Px[mid];
```

```
    // Vencer (Recursin)
    double dl = closestUtil(Px, mid);
    double dr = closestUtil(Px + mid, n - mid); // (Usa aritmética de punteros)

    double d = min(dl, dr);

    // Combinar (Construir la franja)
    struct Point* strip = (struct Point*)malloc(n * sizeof(struct Point));
    // ... (manejo de error) ...
    int j = 0;
    int i;
    for ( i = 0; i < n; i++)
        if (fabs(Px[i].x - midPoint.x) < d) // <-- Clave: solo puntos cerca del centro
            strip[j] = Px[i], j++;

    // Encontrar el mnimo en la franja
    double d_strip = stripClosest(strip, j, d);
    free(strip);

    return min(d, d_strip);
}
```

- Puntos Clave:
    - Líneas de "Vencer": dl = closestUtil(...) y dr = closestUtil(...). Estas son las dos llamadas recursivas que resuelven los subproblemas.
    - Línea de "Combinar" (Filtro): if (fabs(Px[i].x - midPoint.x) < d) es crucial. Reduce el problema de $O(n^2)$ a $O(n)$ al solo considerar puntos que *podrían* formar un par más cercano que d.

4. El Paso de Combinación: stripClosest()

Esta función resuelve la parte más difícil: encontrar el par más cercano *dentro de la franja central*.

```
double stripClosest(struct Point strip[], int size, double d) {
    double min_dist = d;

    // Ordenar la franja por 'y'
    qsort(strip, size, sizeof(struct Point), compareY); // <-- ¡Punto muy importante!

    int i, j;
    // Iterar y comparar con los vecinos cercanos
    for ( i = 0; i < size; ++i) {
        // La magia está aquí (el bucle interno NO es O(n))
```

```
        for ( j = i + 1; j < size && (strip[j].y  - strip[i].y) < min_dist; ++j) {
            if (dist(strip[i],  strip[j]) < min_dist)
                min_dist = dist(strip[i],  strip[j]);
        }
    }
    return min_dist;
}
```

- Puntos Clave:
    - qsort(..., compareY): En *esta implementación*, la franja se ordena por y *cada vez* que se combina. Como la franja puede tener hasta $O(n)$ puntos, este qsort toma $O(n \log n)$ tiempo. Esto hace que el paso de "Combinar" sea $O(n \log n)$ y la complejidad total del algoritmo sea $O(n (\log n)^2)$.
    - Optimización del Bucle Interno: El bucle for (j = ...) no es $O(n)$. La condición (strip[j].y - strip[i].y) < min_dist garantiza (por una prueba geométrica) que este bucle interno se ejecuta un número constante de veces (máx. 6-8 veces). Por lo tanto, el doble bucle *después* del qsort es en realidad $O(n)$, no $O(n^2)$.
    - *(Nota para tu reporte):* Existe una implementación más compleja pero más rápida ($O(n \log n)$ total) donde los puntos se pre-ordenan por y *una sola vez* al inicio, y en lugar de qsort aquí, se usa un algoritmo de "merge" de $O(n)$. Tu código usa la versión más simple de implementar.