1. Investigación

1.1. Ejemplo de usos del comando Time y explicar los 3 tiempos que se están obteniendo

```
#include <stdio.h>
#include <sys/times.h>
#include <unistd.h>
#include <time.h>

int main() {
    struct tms inicio, fin;
    clock_t inicio_real, fin_real;
    long ticks_por_segundo = sysconf(_SC_CLK_TCK);

    inicio_real = times(&inicio);

    for (long i = 0; i < 100000000; i++) {
        // Operacion intensiva en CPU
        double x = 3.14159 * i * i;
    }

    sleep(1);

    fin_real = times(&fin);

    double real = (double)(fin_real - inicio_real) / ticks_por_segundo;
    double user = (double)(fin.tms_utime - inicio.tms_utime) / ticks_por_segundo;
    double sys  = (double)(fin.tms_stime - inicio.tms_stime) / ticks_por_segundo;

    // Resultados
    printf("\nAnalisis de tiempos:\n");
    printf("Real (total):   %.3f s\n", real);
    printf("Usuario (user): %.3f s\n", user);
    printf("Sistema (sys):  %.3f s\n", sys);
    printf("CPU total:      %.3f s (user + sys)\n", user + sys);

    return 0;
}
```

1.2. Explicación de los tres tiempos

1.2.1. Tiempo Real (real / wall clock time)

- Tiempo total transcurrido desde el inicio hasta el final del programa
- Incluye todo: tiempo de ejecución, tiempo de espera, I/O

- Medido con `times()` al inicio y final del proceso

1.2.2. Tiempo de Usuario (user CPU time)

- Tiempo que la CPU ejecutó código del usuario (tu programa)
- Incluye operaciones de cálculo, procesamiento en memoria
- No incluye tiempo de llamadas al sistema ni E/S
- Se obtiene de `tms_utime` en la estructura `tms`

1.2.3. Tiempo de Sistema (sys CPU time)

- Tiempo que la CPU ejecutó llamadas al sistema en nombre de tu programa
- Incluye operaciones como: manejo de memoria, I/O con el kernel
- Se obtiene de `tms_stime` en la estructura `tms`

1.3. Opciones de optimización del compilador GCC

1.3.1. Niveles generales de optimización

- `gcc -O0`: Sin optimización (por defecto, compilación rápida)
- `gcc -O1`: Optimización básica (reduce tamaño y tiempo de ejecución)
- `gcc -O2`: Optimización avanzada (incluye todas -O1 + optimizaciones de velocidad)
- `gcc -O3`: Optimización agresiva (añade vectorización, inline más agresivo)
- `gcc -Os`: Optimiza para tamaño de código (-O2 pero evitando expansiones que aumenten tamaño)
- `gcc -Ofast`: Máxima optimización (incluye -O3 + optimizaciones que violan estándares)

1.3.2. Optimizaciones específicas

- **Vectorización**
  - `gcc -ftree-vectorize`: Habilita vectorización automática (activada en -O2, -O3)
  - `gcc -fopt-info-vec-all`: Muestra información detallada de vectorización
  - `gcc -march=native`: Genera código para la arquitectura local
- **Loop Unrolling**

- `gcc -funroll-loops`: Desenrolla bucles pequeños (activado en -O2, -O3)
- `gcc -funroll-all-loops`: Desenrolla todos los bucles

- **Inline de funciones**
  - `gcc -finline-functions`: Integra funciones pequeñas en el llamador
  - `gcc -finline-limit=N`: Establece límite de tamaño para inline

1.3.3. Optimizaciones para arquitectura

- `gcc -mavx2`: Usa instrucciones AVX2 (para CPUs modernas Intel/AMD)
- `gcc -msse4.2`: Habilita SSE4.2
- `gcc -mtune=haswell`: Optimiza para microarquitectura específica

1.3.4. Optimizaciones de perfilado (PGO)

- `gcc -fprofile-generate`: Genera código instrumentado para perfilado
- `gcc -fprofile-use`: Usa datos de perfilado para optimización guiada

1.3.5. Control de optimizaciones

- `gcc -fno-strict-aliasing`: Desactiva alias estricto (útil para código legacy)
- `gcc -fomit-frame-pointer`: Elimina frame pointer (mejor rendimiento)
- `gcc -pipe`: Usa pipes en lugar de archivos temporales