Análisis empírico y asintótico

Objetivos

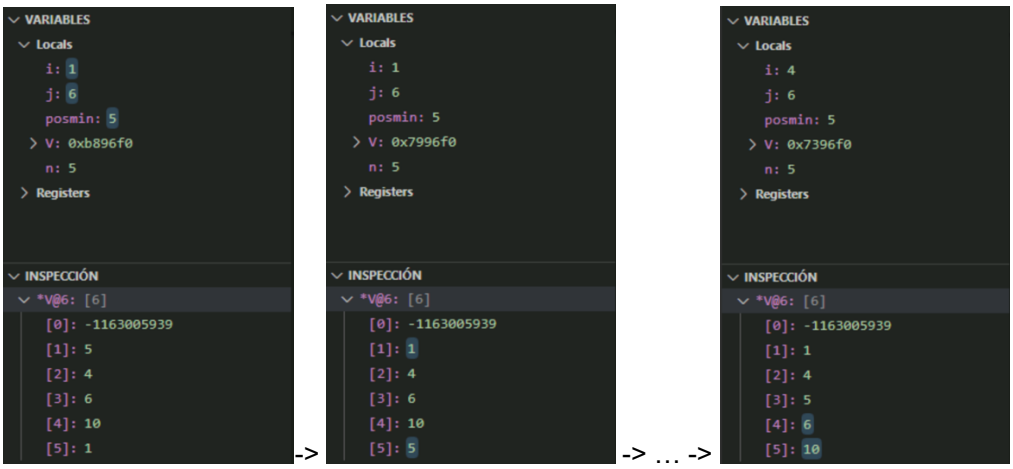
- Realizar trazas de programas usando el depurador GDB y compara con trazas basadas en cout.
- Analizar el tiempo de ejecución empírico de diversos algoritmos utilizando gráficos.
- Analizar las tasas de crecimiento de la complejidad asintótica de diversos algoritmos utilizando gráficos.
- Implementar algoritmos recursivos en C++.
- Trabajar en grupo.

Enunciado

Actividad 1: uso de GDB (parte 2: expresiones y resaltados)

Durante la sesión de prácticas se realizará un ejemplo guiado en el que deberéis utilizar el código implementado en el Práctica 2 para estudiar y poner en práctica alguna funcionalidad adicional del depurador GDB, el cual ya fue introducido en la práctica anterior.

Por ejemplo, se mostrará como utilizar comandos del depurador para mostrar el contenido del vector, y como interpretar los resaltados como los que se muestran en las siguientes imágenes:

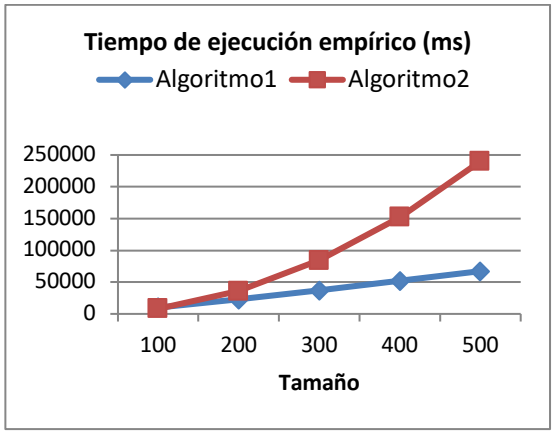


Actividad 2: análisis asintótico (Grupos de 2-3 alumnos)

Durante la sesión, se realizará un análisis comparativo de los tiempos de ejecución empírico y asintótico del comportamiento del algoritmo para diversos valores del tamaño del problema. Se deberán mostrar los resultados en un gráfico tal y como se indica a continuación, y comparar los resultados con los de tus compañeros. Entre todos, comentar las conclusiones que obteneis de los resultados empíricos y asintóticos.

Caso empírico: Representa los valores de los tres casos: caso cualquiera, caso mejor y caso peor.

En la Figura 1 se muestra un ejemplo de un gráfico con los tiempos de ejecución empíricos para dos supuestos algoritmos llamados Algoritmo1 y Algoritmo2. En el eje horizontal se indica el tamaño del problema y en el vertical el tiempo de ejecución en milisegundos.



Como el tiempo de ejecución depende de factores externos hay que tener en cuenta cuáles son las características hardware del equipo físico (procesador, memoria RAM…) y el software (sistema operativo, lenguaje de programación, compilador...), en el que se realizan los experimentos.

En esta actividad el gráfico contendrá tres líneas, una para cada caso analizado del algoritmo.

Figura 1: Tiempo de ejecución empírico.

Caso asintótico: Representa las complejidades asintóticas de los casos mejor y peor.

En la Figura 2 se muestra, como ejemplo, un gráfico con las tasas de crecimiento de los algoritmos Algoritmo1 y Algoritmo2, suponiendo que sus órdenes asintóticos son O(logn) y O(n^2), respectivamente.

En el eje horizontal se indica el tamaño y en el vertical el valor del orden.

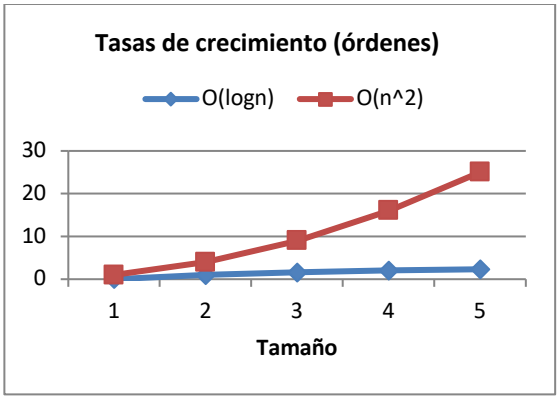


Figura 2: Tasas de crecimiento.

Actividad 3: recursividad y GDB (parte 3: la pila de llamadas)

Las combinaciones sin repetición de n elementos cogidos de k en k, $C_{n,k}$, son el número total de grupos distintos de k elementos que se pueden formar a partir de un total de n elementos (k≤n).

Ejemplo:

Si queremos distribuir n=4 personas (Antonio, José, María y Carmen) en grupos de k=3 personas, tenemos un total de $C_{4,3} = 4$ formas distintas de distribuirlas. En concreto, los grupos que se pueden formar son:

- Grupo 1: Antonio, José, María
- Grupo 2: Antonio, José, Carmen
- Grupo 3: Antonio, María, Carmen
- Grupo 4: José, María, Carmen

El valor $C_{n,k}$ se puede calcular recursivamente de la siguiente forma

$$C_{n,k} = \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ C_{n-1,k-1} + C_{n-1,k} & \text{en caso contrario} \end{cases}$$

a) Durante la sesión se implementará en C++ una función llamada **combinaciones** correspondiente al pseudocódigo anterior y se creará un programa que pida por teclado los valores de n y k, realice una llamada a la función e imprima el número total de combinaciones que devuelve la función.

b) A continuación, se muestra un ejemplo de una traza manual que contiene errores para un ejemplo concreto con n=4 y k=3. Ten en cuenta que al inicio de cada línea aparece el número de llamada recursiva: "1.-", "2.-"", 3.-, etc. y a continuación el nombre de la función junto con el valor de los parámetros n y k de cada llamada. Por otro lado, las llamadas recursivas que se realicen desde una función se imprimirán un nivel más adentro.

Con errores:	Sin errores:
1.- combinaciones_traza(4,3) 2.- combinaciones_traza(3,2) 4.- combinaciones_traza(1,0) 5.- combinaciones_traza(2,1) 7.- combinaciones_traza(1,1) 8.- combinaciones_traza(2,2) 3.- combinaciones_traza(3,3)	¿?¿?

Identifica y corrige los errores de la traza anterior. Utiliza el depurador GDB para hacer un seguimiento de la pila de llamadas recursivas que se producen al ejecutar el programa para dicho ejemplo.

Por último, implementa una nueva función llamada **combinaciones_traza** que reproduzca la traza anterior mediante cout.