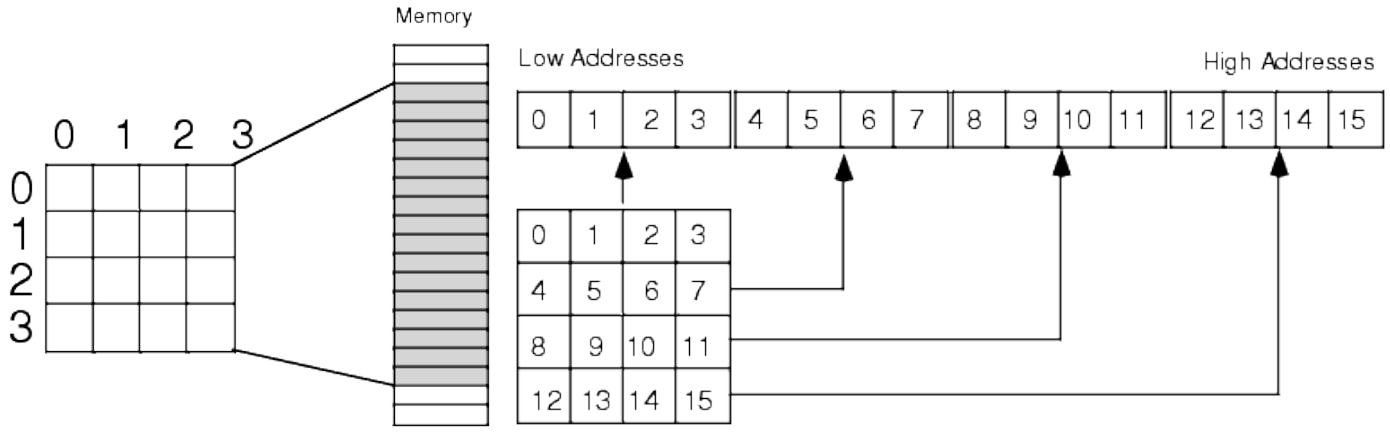
- Crear un programa en C que multiplique 2 matrices cuadradas y el resultado lo almacene en una tercera matriz.
- Posteriormente variar el orden de los bucles *for* en sus 6 combinaciones para cambiar la forma de acceso a la información de la estructura de datos.
- Cronometrar el tiempo que tarda cada una en resolverse.
- Ejecutar al menos 3 diferentes sistemas de matrices.
- Compara los resultados y explica cuál disposición de ciclos *for* se ejecuta más rápido y cuál más lento. ¿Por qué ocurre esto?

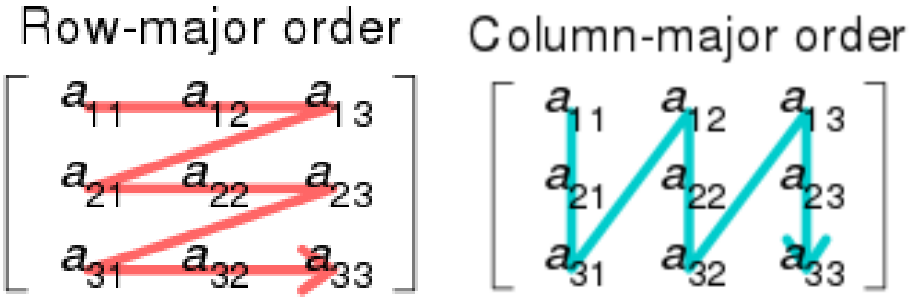
**Respuestas.**

En 5 de los 6 tamaños de matrices se logra observar que la combinación de "*kij*" es la que termina mucho más rápido que las demás, en la única en la que no gana es cuando la matriz es pequeña, mientras que la que tarda demasiado con respecto a los demás hay una situación curiosa, ya que durante las primeras 4 multiplicaciones la combinación "*kji*" pero la combinación "*jki*" es la que la sigue muy de cerca siendo que los tiempos que ambos tardan es mínimo. El que uno termine más rápido que otro se debe a la forma de acceso de almacenamiento de la matriz la cual hace un barrido a las matrices de manera *Row-major order*.

Para entender mejor esa parte primero debemos de mencionar la forma en la que se guardan los datos en la memoria, nosotros al pensar en una matriz pensamos en algo bidimensional, pero en memoria eso no pasa debido a que ésta está en forma de vector, lo que hace que por ejemplo el elemento [1][1] en forma matricial podemos ver que a su alrededor tiene como vecinos a los elementos [1][0], [1][2],[0][1],[2][1] por lo que leer cualquiera de esos datos debería de tomar el mismo tiempo, ¿no?, la realidad es que no debido a la forma de almacenamiento en memoria es en forma de arreglo como se mencionó, siguiendo con el ejemplo vemos que en realidad el elemento [1][1] solo tiene como vecinos a [1][0] y a [1][2] por lo que acceder a los demás elementos tomará más tiempo.



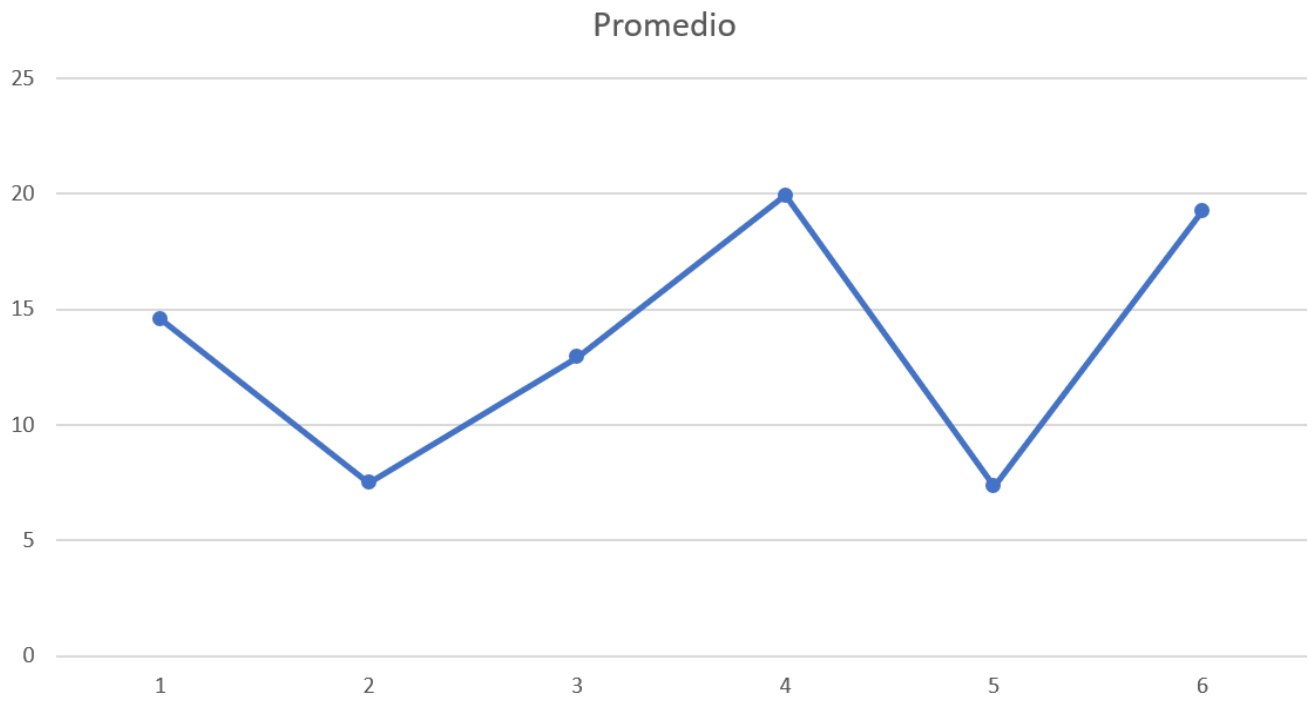

Por lo que pasando al barrido que se le puede dar a las matrices tenemos dos tipos, uno es el *Row-major order* y el otro es el *Column-major order*

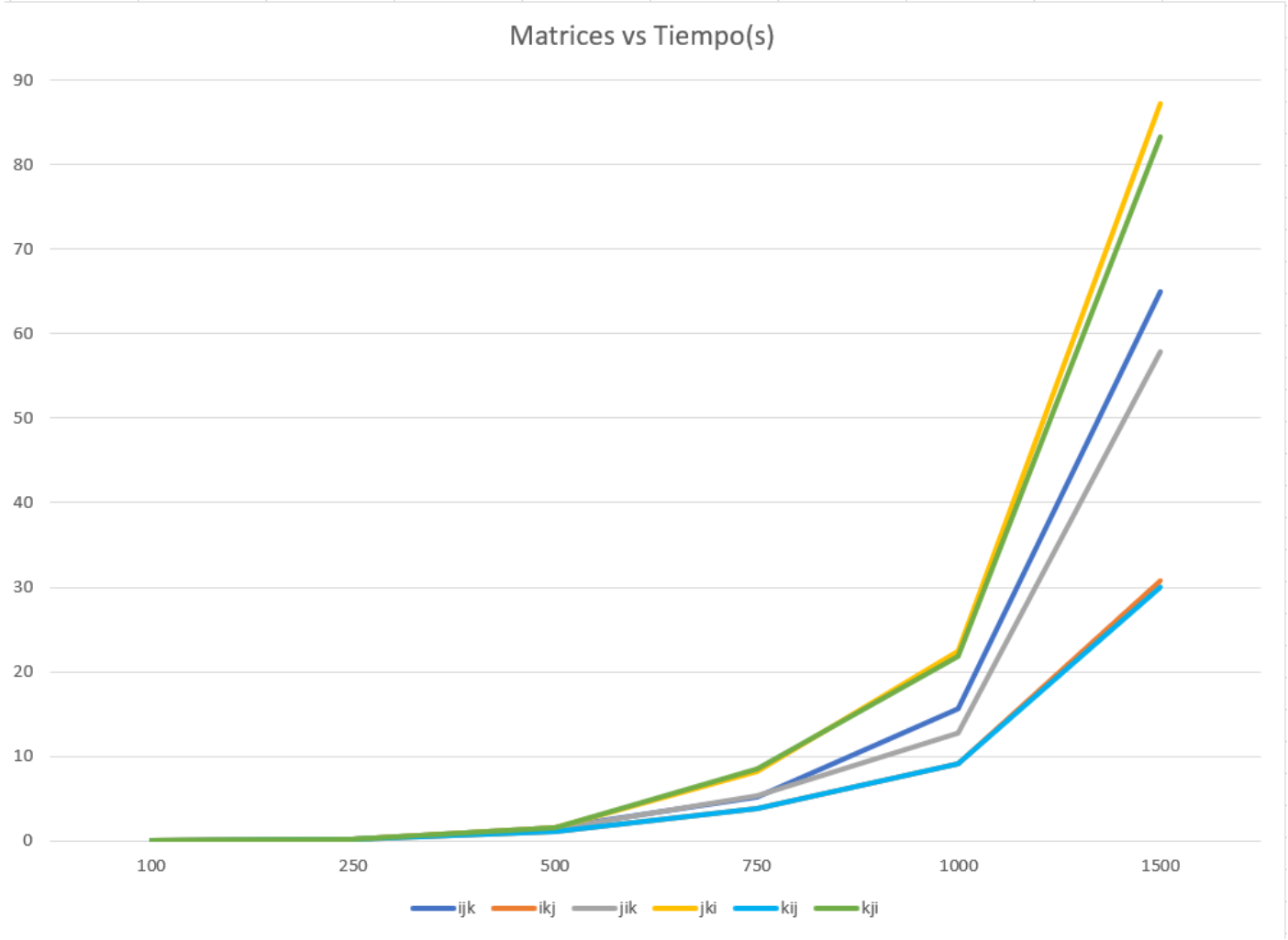



Vemos que el *Row-major order* recorre la matriz en renglones lo que hace que al avanzar de un elemento a otro sea rápido debido a que a lo explicado anteriormente a la forma de almacenar una matriz en memoria. Mientras en el caso del *Column-major order* hace el barrido por columnas lo que basandonos en la memoria se tiene que hacer demasiados saltos para ir de un elemento a otro.

Por lo que para realizar la multiplicación de matrices la forma más eficas de realizarla es que se realice el barrido de ambas matrices por *Row-major order* por los tiempos que toma en los saltos en la memoria.

| | 100 | 250 | 500 | 750 | 1000 | 1500 | Promedio | |
|---|---|---|---|---|---|---|---|---|
| ijk | 0.010971 | 0.186721 | 1.515536 | 5.233959 | 15.607469 | 64.951541 | 14.5843662 | 1 |
| ikj | 0.013551 | 0.143158 | 1.133716 | 3.828459 | 9.177686 | 30.726997 | 7.50392783 | 2 |
| jik | 0.012662 | 0.166402 | 1.459458 | 5.304476 | 12.779353 | 57.896158 | 12.9364182 | 3 |
| jki | 0.013628 | 0.189059 | 1.543512 | 8.217262 | 22.415635 | 87.238879 | 19.9363292 | 4 |
| kij | 0.012336 | 0.141579 | 1.111752 | 3.767775 | 9.083389 | 29.968273 | 7.34751733 | 5 |
| kji | 0.013696 | 0.192081 | 1.548494 | 8.540695 | 21.892335 | 83.282914 | 19.2450358 | 6 |
| Promedio | 0.01280733 | 0.16983333 | 1.38541133 | 5.81543767 | 15.1593112 | 59.0107937 | | |






En conclusión con las graficas mostradas sacadas de los tiempos de cada multiplicación con cada combinación posible de los ciclos for, vemos que en teoría por lo explicado arriba el que debería de ser más rápido en terminar debe de ser la combinación "*i k j*" pero en la práctica nos quedó que el más rápido fue "*k i j*" pero por muy poco tiempo por lo que podemos decir que si cumple. Mientras que en el caso del más lento fue la combinación "*jki*"dado que el barrimiento que utiliza es el *Column-major order*.

## Código

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

void get_walltime(double* wcTime) {
    struct timeval tp;
    gettimeofday(&tp, NULL);
    *wcTime = (tp.tv_sec + tp.tv_usec/1000000.0);

}

int main(int argc, char* argv[]) {
    //Variables para los ciclos for, y el tamano de las matrices
    int i, j,k,n;
    //Variables extras
    int a,tamano[6] = {100,250,500,750,1000,1500};
    //Creacion de las matrices
    int **matrizA, **matrizB, **matrizC;
    //Variables para el calculo de tiempo
    double S1,E1;

    for(a = 0; a < 6; a++){
        n = tamano[a];
        printf("Matriz de %d * %d\n",n,n);
        //Inicializando matrices
        matrizA = (int **)malloc(n * sizeof(int *));
        matrizB = (int **)malloc(n * sizeof(int *));
        matrizC = (int **)malloc(n * sizeof(int *));
        for (i = 0; i < n; i++) {
            *(matrizA + i) = (int *)malloc(n * sizeof(int *));
            *(matrizB + i) = (int *)malloc(n * sizeof(int *));
            *(matrizC + i) = (int *)malloc(n * sizeof(int *));
        }
        //Llenado de matrices
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                matrizA[i][j] = rand() % 6;
                matrizB[i][j] = rand() % 6;
            }
        }
```

```
41
42          printf("Terminé de llenar las matrices :)\n");
43
44          //Primera combinacion
45          get_walltime(&S1);
46          for (i = 0; i < n; i++) {
47              for (j = 0; j < n; j++) {
48                  for (k = 0; k < n; k++) {
49                      matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
50                  }
51              }
52          }
53          get_walltime(&E1);
54          printf("1)Tiempo método ijk: %f s\n", (E1 - S1));
55
56          matrizC = (int **)malloc(n * sizeof(int *));
57          for (i = 0; i < n; i++) {
58              *(matrizC + i) = (int *)malloc(n * sizeof(int *));
59          }
60
61          //Segunda combinacion
62          get_walltime(&S1);
63          for (i = 0; i < n; i++) {
64              for (k = 0; k < n; k++) {
65                  for (j = 0; j < n; j++) {
66                      matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
67                  }
68              }
69          }
70          get_walltime(&E1);
71          printf("2)Tiempo método ikj: %f s\n", (E1 - S1));
72
73          matrizC = (int **)malloc(n * sizeof(int *));
74          for (i = 0; i < n; i++) {
75              *(matrizC + i) = (int *)malloc(n * sizeof(int *));
76          }
77
78          //Tercera combinacion
79          get_walltime(&S1);
80          for (j = 0; j < n; j++) {
```

```
81              for (i = 0; i < n; i++) {
82                  for (k = 0; k < n; k++) {
83                      matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
84                  }
85              }
86          }
87          get_walltime(&E1);
88          printf("3)Tiempo método jik: %f s\n", (E1 - S1));
89
90          matrizC = (int **)malloc(n * sizeof(int *));
91          for (i = 0; i < n; i++) {
92              *(matrizC + i) = (int *)malloc(n * sizeof(int *));
93          }
94
95          //Cuarta combinacion
96          get_walltime(&S1);
97          for (j = 0; j < n; j++) {
98              for (k = 0; k < n; k++) {
99                  for (i = 0; i < n; i++) {
100                     matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
101                 }
102             }
103         }
104         get_walltime(&E1);
105         printf("4)Tiempo método jki: %f s\n", (E1 - S1));
106
107         matrizC = (int **)malloc(n * sizeof(int *));
108         for (i = 0; i < n; i++) {
109             *(matrizC + i) = (int *)malloc(n * sizeof(int *));
110         }
111
112         //Quinta combinacion
113         get_walltime(&S1);
114         for (k = 0; k < n; k++) {
115             for (i = 0; i < n; i++) {
116                 for (j = 0; j < n; j++) {
117                     matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
118                 }
119             }
120         }
```

```
121         get_walltime(&E1);
122         printf("5)Tiempo método kij: %f s\n", (E1 - S1));
123 
124         matrizC = (int **)malloc(n * sizeof(int *));
125         for (i = 0; i < n; i++) {
126             *(matrizC + i) = (int *)malloc(n * sizeof(int *));
127         }
128 
129         //Sexta combinacion
130         get_walltime(&S1);
131         for (k = 0; k < n; k++) {
132             for (j = 0; j < n; j++) {
133                 for (i = 0; i < n; i++) {
134                     matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
135                 }
136             }
137         }
138         get_walltime(&E1);
139         printf("6)Tiempo método kji: %f s\n", (E1 - S1));
140     }
141 
142     return 0;
143 }
```

## Resultados

```
ramy@Ramy:~$ ./mpi
Matriz de 100 * 100
Terminé de llenar las matrices :)
1)Tiempo método ijk: 0.010971 s
2)Tiempo método ikj: 0.013551 s
3)Tiempo método jik: 0.012662 s
4)Tiempo método jki: 0.013628 s
5)Tiempo método kij: 0.012336 s
6)Tiempo método kji: 0.013696 s
Matriz de 250 * 250
Terminé de llenar las matrices :)
1)Tiempo método ijk: 0.186721 s
2)Tiempo método ikj: 0.143158 s
3)Tiempo método jik: 0.166402 s
4)Tiempo método jki: 0.189059 s
5)Tiempo método kij: 0.141579 s
6)Tiempo método kji: 0.192081 s
Matriz de 500 * 500
Terminé de llenar las matrices :)
1)Tiempo método ijk: 1.515536 s
2)Tiempo método ikj: 1.133716 s
3)Tiempo método jik: 1.459458 s
4)Tiempo método jki: 1.543512 s
5)Tiempo método kij: 1.111752 s
6)Tiempo método kji: 1.548494 s
Matriz de 750 * 750
Terminé de llenar las matrices :)
1)Tiempo método ijk: 5.233959 s
2)Tiempo método ikj: 3.828459 s
3)Tiempo método jik: 5.304476 s
4)Tiempo método jki: 8.217262 s
5)Tiempo método kij: 3.767775 s
6)Tiempo método kji: 8.540695 s
Matriz de 1000 * 1000
Terminé de llenar las matrices :)
1)Tiempo método ijk: 15.607469 s
2)Tiempo método ikj: 9.177686 s
3)Tiempo método jik: 12.779353 s
4)Tiempo método jki: 22.415635 s
5)Tiempo método kij: 9.083389 s
6)Tiempo método kji: 21.892335 s
Matriz de 1500 * 1500
Terminé de llenar las matrices :)
1)Tiempo método ijk: 64.951541 s
2)Tiempo método ikj: 30.729697 s
3)Tiempo método jik: 57.896158 s
4)Tiempo método jki: 87.238879 s
5)Tiempo método kij: 29.968273 s
6)Tiempo método kji: 83.282914 s
```