Pregunta 1

El problema del viajante o vendedor viajero responde a la siguiente pregunta: dadas *n+1* ciudades (enumeradas de 0 a *n*) y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que inicia en la ciudad 0, visita cada ciudad una vez y al finalizar regresa a la ciudad 0? La siguientes funciones resuelven este problema secuencialmente en tiempo *O((n+1)!)*:

```
int viajante(int z[], int n,
             int **m) {
  int x[n+1];
  for (int k= 0; k<=n; k++)
    x[k]= k;
  int min= INT_MAX;
  // ciclo de búsqueda
  for (int k= 1; k<=n; k++) {
    swap(x, 1, k); // x[1] es k
    // genera todas las permutaciones
    // de x[2] ... x[n]
    perms(x, 2, n, m, z, &min);
    // restaura x a su estado inicial
    swap(x, 1, k);
  }
  return min;
}
```

```
void perms(int x[], int i, int n,
     int **m, int z[], int *pmin) {
  if (i<n) {
    for (int k= i; k<=n; k++) {
      swap(x, i, k);
      // permutaciones de x[i+1] ... x[n]
      perms(x, i+1, n, m, z, pmin);
      swap(x, i, k);
  } }
  else { // x es una permutacion
    int d= dist(x, n, m);
    if (d<*pmin) {
      *pmin= d;
      for (int k=0; k<=n; k++)
        z[k]= x[k];
} } }
```

La solución se obtiene invocando la función *viajante* que recibe como parámetros *n*, una matriz *m* con las distancias entre cada par de ciudades y un arreglo *z* en donde se entregará la ruta más corta. Esta última será la que pasa por las ciudades *0*, *z[1]*, *z[2]*, …, *z[n]*, *0*. Para calcularla, la función *viajante* coloca en el arreglo *x* las ciudades *0* a *n* en ese orden y hace un ciclo de búsqueda. En la *k*-ésima iteración intercambia en *x* la posición de las ciudades *1* y *k* e invoca *perms* para generar las *(n-1)!* permutaciones de las ciudades restantes en *x[2]* a *x[n]*. Cada permutación en *x* representa una ruta. En *perms*, la función dada *dist* usa la matriz *m* para calcular la distancia total recorrida por una ruta. Si esta distancia es más corta que la que almacena *min*, *perms* actualiza *min* con esta nueva distancia y coloca la ruta en el arreglo *z*. No necesitará modificar ni entender como funciona *perms*.

Reprograme la función *viajante* para que la búsqueda de la ruta más corta se haga en paralelo en *n* cores. Para ello use *fork* para hacer que cada iteración del ciclo de búsqueda se ejecute en un proceso pesado. El *k*-ésimo proceso se encarga de calcular la ruta más corta para el caso en que *x[1]* es *k*. Lo hace llamando a *perms* para generar las permutaciones del resto de las ciudades, dejando en *z* la ruta más corta y en *min* la distancia recorrida. Use un pipe por cada proceso hijo para que este entregue *min* y *z* al padre. El padre calcula la más corta de las *n* rutas calculadas por los hijos almacenándola en *z*.

Revise que su solución posea paralelismo. Si no, su nota será muy baja.

Pregunta 2

Esta pregunta consiste en paralelizar el cálculo numérico de $\int_0^1 f(x)dx$. Para ello se descompone el intervalo [0,1] en 1000 subintervalos de la forma $[i*\Delta x,(i+1)*\Delta x]$ con $\Delta x=\frac{1}{1000}$ e *i* tomando valores entre 0 y 999. Para esta paralelización se utiliza un número desconocido de computadores single-core conectados en red y que actuarán como clientes.

Un proceso servidor corre en anakena (comando *./coordinador*) y se encarga de enviar subintervalos a los clientes y mostrar el resultado final. El servidor acepta conexiones de los clientes a través del puerto 3000 y crea para cada uno de ellos un thread que se encarga de enviar un subintervalo a ese cliente, esperar la recepción de la integral parcial, enviar de inmedianto un nuevo subintervalo, y así hasta que se acaben los 1000 subintervalos. Entonces muestra el resultado final como la suma de las integrales parciales.

Cada cliente (comando *./integrar*) calcula numéricamente la integral parcial de *f* en cada subintervalo *[xi, xf]* recibido del servidor. Esto lo hace llamando a la función dada *double integral_f(double xi, double xf)*. Esto toma tiempo porque requiere evaluar *f* en muchísimos puntos. Después envía el resultado al servidor, recibe de inmediato un nuevo subintervalo (si aún quedan) y continúa calculando, sin permanecer ocioso en ningún momento.

El siguiente es un ejemplo de uso que muestra el servidor trabajando con 3 clientes. Los clientes pueden llegar en cualquier momento. El despliegue de los comandos se muestra en orden cronológico.

servidor	*cliente 1*	*cliente 2*	*cliente 3*
`$ ./coordinador`			
	`$ ./integrar`		
`env [0.000, 0.001]`	`rec [0.000, 0.001]`	`% ./integrar`	
`env [0.001, 0.002]`		`rec [0.001, 0.002]`	
`env [0.002, 0.003]`	`rec [0.002, 0.003]`		`% ./integrar`
`env [0.003, 0.004]`			`rec [0.003, 0.004]`
`env [0.004, 0.005]`		`rec [0.004, 0.005]`	
`env [0.005, 0.006]`			`rec [0.005, 0.006]`
`env [0.006, 0.007]`	`rec [0.006, 0.007]`		
…	…	…	…
`env [0.999, 1.000]`		`rec [0.999, 1.000]`	
`integral= 5.30450`	`$`	`$`	`$`

Programe el servidor (*coordinador*) y el cliente (*integrar*) de manera que reproduzcan exactamente las salidas estándares mostradas en el ejemplo de uso. En el servidor no programe la función *main*, solo programe la función *serv*. No se preocupe por el término del servidor. Sí debe preocuparse por el término de los clientes.