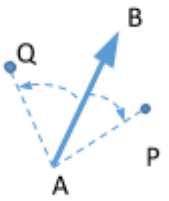
1. Caída de tensión en una malla

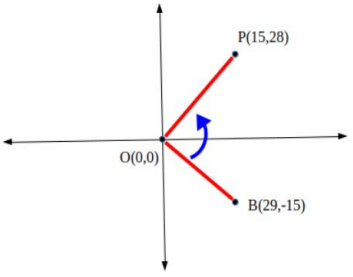
Se tiene un circuito eléctrico de **n** componentes conectados en serie y en forma circular (**malla**). Son conocidas las variaciones de potencial en cada componente. Por tratarse de una malla, cumplen la Ley de Kirchhoff que establece que la suma de los voltajes en una malla es cero. Se desea saber entre que componentes se puede medir la mayor caída neta de potencial. Por ejemplo, si las variaciones (voltajes) fueran {1, -3, 5, 2, 2, -8, 1} la mayor caída de potencial corresponde a -8 + 1 + 1 - 3 = - 9 que estaría entre los nodos 5 y 1 (nodo 5 + nodo 6 + nodo 0 + nodo 1). Escriba un programa que encuentre la caída de potencial máxima, y entre que nodos se da. La entrada es un vector de tamaño n cuyas componentes representan la caída de potencial en cada nodo.

2. Segmentos y Polígonos en 2D

a- **Dirección de un punto con respecto a un segmento orientado**. Se dice que un punto P está a la derecha de un segmento orientado, si parado sobre el segmento, en la dirección de éste, debemos rotar la mirada en el sentido de las agujas del reloj para mirar a P. En forma análoga, un punto Q está a la izquierda de un segmento orientado, si parado sobre el segmento, en la dirección de éste, debemos rotar la mirada en el sentido contrario a las agujas del reloj para mirar a Q.



El producto cruz posee una propiedad interesante que es utilizada para determinar la dirección de un punto respecto de un segmento orientado: *"El producto cruz entre 2 puntos es positivo (componente z hacia arriba) sí y sólo si el ángulo de esos puntos en el origen de coordenadas es en contra del movimiento de las agujas del reloj, y será negativo (componente z hacia abajo) si el ángulo de esos puntos en el origen es en el sentido de giro de las agujas del reloj, en el caso que sean colineales, es cero (componente z nula)"*. Se solicita hacer una función que determine si un punto está o no a la derecha de un segmento ordenado. Prototipo (Respetarlo):




```
// retorna 1 si 'point' está a la derecha del segmento, -1 si está a la izquierda
// y 0 si son colineales.
int isOnTheRight( P2D_t point, P2D_t startSeg, P2D_t endSeg);
```

El tipo **P2D_t** está definido como:

```
struct  P2D_t{
```

```
    double X;
    double Y;
};
```

b- **Clasificación de un polígono como convexo.** Se dice que un polígono es convexo si todos los segmentos orientados que lo definen cumplen la condición de que todos los puntos de los restantes segmentos están en la misma dirección (o todos están a la derecha, o todos están a la izquierda). Teniendo en cuenta esto, se solicita implementar la función (respetar el prototipo):

```
// Retorna 1 si el polígono recibido es convexo, 0 en caso contrario
// Recibe un polígono de numPoints vértices
int isConvexPolygon( P2D_t polygon[], int numPoints);
```

c- **Punto interior a un polígono.** Un punto interior a un polígono orientado cumple la propiedad que si se toma a éste como centro y se recorren los segmentos orientados en orden entonces el ángulo total barrido es ±2pi (ver [1]). Por el contrario, si el punto es externo, el ángulo barrido es 0. Implementar la función:

```
// Retorna 1 si el punto es interior, 0 en caso contrario
// Recibe un punto y el polígono de numPoints vértices
int isInside( P2D_t point, P2D_t polygon[], int numPoints);
```

[1] El ángulo barrido por un segmento orientado se toma en [-pi, pi].

3. Estadísticas de caracteres

Se desea encontrar la frecuencia de aparición de caracteres alfanuméricos en un texto (sin discriminar mayúsculas de minúsculas), es decir la cantidad de veces que aparece la letra ‘a’ (o ‘A’), ‘b’ (o ‘B’), y así hasta la ‘z’ (o ‘Z’), igualmente para los dígitos (‘0’, ‘1’, …) y contando también otros (todo lo que no sea caracteres alfabéticos ni dígitos).

Hacer un programa que encuentre estas frecuencias de aparición tomando caracteres desde el dispositivo estándar de entrada.

El programa tiene que imprimir antes de finalizar la letra más frecuente, el dígito más frecuente y la cantidad de otros caracteres.

4. Juegos de azar

Se solicita simular dos juegos de azar, basados en el lanzamiento de ciertas monedas. Si sale cara, el jugador ganará 1 peso; y si sale cruz, lo perderá.

El primer juego, llamémosle A, es muy sencillo: consiste en lanzar al aire una moneda en la que la probabilidad de cara es algo inferior a 0.5 (digamos, 0.495). Ese desequilibrio de la moneda, aunque pequeño, hará que el juego no sea recomendable: a largo plazo el jugador tenderá a perder.

a. Implemente el Juego A. Considerando un capital inicial de 10000 pesos encuentre el capital luego de 20000 jugadas.

El segundo juego, llamado B, es más complejo. Antes de jugarlo hay que ver si el capital del jugador –fruto de las ganancias y pérdidas acumuladas– es o no múltiplo de 3. Si no lo es (está, por ejemplo, en 8 pesos), tirará con una moneda "buena", trucada a su favor, cuya probabilidad de cara es casi de 0.75 (digamos, 0.745). Si por el contrario su capital es múltiplo de 3 (el jugador tiene, por ejemplo, 9 pesos), le tocará tirar con una moneda "mala", trucada en su contra, en la que la probabilidad de cara es un poco menor de 0.10 (digamos, 0.0995). Este segundo juego B parece mejor negocio, porque toca jugar con la moneda "buena" con relativa frecuencia: dos de cada tres veces.

Para calcular el resultado del juego B hay que tener presente que la selección de qué moneda se usa cada vez está en función del resultado de las tiradas precedentes. Analizándola matemáticamente resulta que la moneda "mala" acabará usándose más de un tercio de las veces. En suma, como la moneda "mala" se termina usando más de lo que parece, también el juego B arroja pérdidas a largo plazo.

b. Implemente el Juego B. Considerando un capital inicial de 10000 pesos encuentre el capital luego de 20000 jugadas.

Curiosidad: Considerados de forma aislada, tanto el juego A como el B arrojarán pérdidas a largo plazo. Lo curioso es que cuando se combinan de cierta forma ambos juegos… ¡el resultado se vuelve favorable para el jugador! Eso ocurre, en particular, cuando se juegan en secuencias de dos en dos –esto es siguiendo el patrón AABBAABB…

Si quiere verificarlo (NO SE PIDE PARA EL PARCIAL) considere un capital inicial de 100 pesos y encuentre el capital luego de 5000 jugadas compuestas, donde cada jugada consiste en jugar dos veces al juego A y luego dos veces al juego B.

5. Simulación de un cardumen

Es posible simular el comportamiento de cardumen con tan solo 3 reglas muy simples.

1) Tender a acercarse al centro ($rc = \frac{\sum_j^{N_p} r_j}{N_p}$) del cardumen $\delta v1_i = \frac{rc - r_i}{8}$
2) Evitar chocar con los vecinos más cercanos $\delta v2_i = \sum_j^{|r_j - r_i|<3} \frac{r_i - r_j}{|r_j - r_i|}$ (solo se tienen en cuenta los peces que están a una distancia menor a 3 (**`MAX_DISTANCIA_R2`**))
3) Tender a igualar la velocidad media ($vc = \frac{\sum_j^{N_p} v_j}{N_p}$) del cardumen $\delta v3_i = \frac{vc - v_i}{8}$

El cambio en la velocidad del pez i-ésimo será la suma de los cambios debido a estas 3 reglas $\delta v_i = \delta v1_i + \delta v2_i + \delta v3_i$ y la velocidad de cada pez no puede superar la velocidad máxima (4). Con esta velocidad se calcula la evolución temporal de la posición $r_i(t + \delta t) = r_i(t) + v_i(t) \times \delta t$

Por simplicidad vamos a limitarnos a 2D, los pasos temporales son de a 1 unidad y representaremos a nuestras entidades con las siguientes estructuras y definiciones:

```
#define MAX_VEL           4
#define MAX_PECES        16
```

```
#define MAX_DISTANCIA_R2  3

struct V2_t {
    float x, y;
}; // posición o velocidad en 2D

struct Pez_t {
    V2_t pos;
    V2_t vel;
}; // representa un pez con su posición y velocidad en 2D
```

En el programa, un cardumen será representado por un arreglo de peces (**`Pez_t cardumen[MAX_PECES]`**). Para realizar la simulación se deben implementar las siguientes funciones:

```
/* inicializa los primeros npeces elementos del arreglo cardumen con
posiciones al azar dentro de un área cuadrada de n x n y con
velocidades al azar con 0 < |v| < MAX_VEL. */
void init_cardumen(Pez_t cardumen[], int npeces, int n);

/* Calcula el cambio de velocidad que debería realizar el pez ip
debido a la regla 1 */
V2_t regla1(Pez_t viejo_cardumen[],int np, int ip);

/* Calcula el cambio de velocidad que debería realizar el pez ip
debido a la regla 2 */
V2_t regla2(Pez_t viejo_cardumen[],int np, int ip);

/* Calcula el cambio de velocidad que debería realizar el pez ip
debido a la regla 3 */
V2_t regla3(Pez_t viejo_cardumen[],int np, int ip);

/* aplica las 3 reglas para calcular el cambio de velocidad de cada
pez en el cardumen y luego actualiza las posiciones con estas nuevas
velocidades */
void mover_peces(Pez_t cardumen[],int np);

/* funcion para limitar la velocidad. Si |v| > MAX_VEL v=
(v/|v|)*MAX_VEL. Se utiliza en mover_peces para que las velocidades
sean siempre razonables */
V2_t limitar_velocidad(V2_t v);

/* imprime el estado actual del cardumen. Nada sofisticado, sólo las
posiciones y velocidades de cada pez para poder inspeccionar la
evolución temporal a ojito */
void print_cardumen(Pez_t cardumen[], int np, int tiempo);
```

Nota 1: puede resultar útil (haciendo el código más claro) si también se implementan funciones para sumar, restar, multiplicar por un escalar y calcular el módulo de los vectores `V2_t`;

Nota 2: Tenga en cuenta que las 3 reglas descriptas, las posiciones (r_j) y velocidades (v_j) están representados por un **`V2_t`**, no por escalares.

6. Números de Uber.

Se dice que un número es el enésimo **número Uber** $T(n)$ si es el menor número que se puede descomponer como n sumas distintas de dos cubos positivos.

Se desea conocer (no usando la WEB) los valores de *T(1)*, *T(2)* y *T(3)*.

Una forma de hacerlo (costosa, pero viable para este orden de números) es armar una matriz *V* de 500x500 en donde:

$$V[i][j] = i^3 + j^3 \qquad para\ i > 0\ y\ j > 0$$

Si después se pasan los valores de la matriz a un vector en donde cada elemento del vector mantenga la terna { i, j, V[*i*][*j*] } y se ordena en forma creciente de acuerdo a V[*i*][*j*], la primer secuencia igual (respecto a V[*i*][*j*]) de largo 1 definirá *T(1)*, la primer secuencia igual de largo 2 definirá *T(2)* y la primer secuencia igual de largo 3 definirá *T(3)*.

Para el vector se sugiere usar elementos del tipo:

```
struct Terna_t{
    int i;
    int j;
    int v;
};
```

Se solicita calcular *T(1)*, *T(2)* y *T(3)*, así como también las sumas que los definen.

Nota: Se podría omitir el armado de la matriz y generar directamente el vector. Asimismo, en el vector, podría omitirse V[*i*][*j*] y calcularlo cada vez que haga falta.

Estimación de quiebre.

Se desea ajustar con dos rectas una serie de datos como los que se muestran en las figuras. El ajuste de las dos rectas en simultáneo debe minimizar la función $\mathbf{Chi^2(a_1,b_1,a_2,b_2)=Chi_1^2(a_1,b_1)+Chi_2^2(a_2,b_2)}$, donde $\mathbf{a_i}$, $\mathbf{b_i}$, $\mathbf{Chi_i}$ son la ordenada, la pendiente y la función Chi cuadrado del ajuste de la recta i . La intersección de las dos rectas es un estimador del punto de quiebre de los datos.

Implementar una función **`estimaQuiebre`** que retorne el índice del dato en el que se produce el quiebre.

En el archivo 'regresion_lineal.h' se encuentra declarada la estructura **`FitParams`** y la función **`fitLineal`** (implementada en 'regresión_lineal.c'):

```
struct FitParams {
    double a, b, siga, sigb, chi2,sigdat;
};

FitParams fitLineal(int ndata, double x[], double y[]);
```

La función **`fitLineal`** ajusta una recta y=a+b*x a los **`ndata`** puntos formados por los pares **`x[i]`**, **`y[i]`** y retorna la estructura de parámetros que caracterizan el ajuste. En particular nos interesa el $\mathbf{Chi^2}$ para encontrar el mínimo de $\mathbf{Chi^2(a_1,b_1,a_2,b_2)}$.

Implementar un programa que lea un archivo con una columna de 256 datos. Estos datos son los valores de la ordenada **`y[i]`** y su índice será el de la abscisa **`x[i]=i`**. Luego aplique la función **`estimaQuiebre`** para encontrar el índice del dato en el que se produce el quiebre. Probar su programa con los archivos de datos que se muestran en la figura.