Pregunta 1

Programe la función *sumaVecs* con el siguiente encabezado:

```
typedef unsigned long long Vector;
Vector sumaVecs(Vector x, Vector y);
```

El tipo *Vector* representa un vector que comprime 8 números enteros de 8 bits sin signo en un solo entero de 64 bits sin signo. Si los bits de *x* son $x_{63}...x_0$, para efectos de esta pregunta nos referiremos al *k*-ésimo entero de *x* como *x[k]* y corresponde a los bits $x_{(k+1)*8-1} \ldots x_{k*8}$, con *k*= 0...7. Por ejemplo si *x* es 0x**ff** 01 67 00 a1 00 02 **03**, *x[0]* es 3 y *x[7]* es 0xff (255). La función *sumaVecs* recibe 2 vectores *x* e *y* y retorna un vector con la suma de ambos vectores. Es decir el resultado *z* es tal que *z[k]=x[k]+y[k]*. Ejemplo de uso:

```
Vector x = 0xff 01 67 01 a1 00 00 03;
Vector y = 0x01 09 03 ff 1f 00 00 ff;
Vector z = sumaVecs(x, y);
//     z = 0x00 0a 6a 00 c0 00 00 02
```

Observe que al sumar los números marcados en negritas en *x* e *y*, se excede el rango representable en 8 bits. En ningún caso este desborde debe afectar el resultado de la suma de la siguiente columna (marcada en negritas en *z*). Es decir el resultado de calcular *x[0] + y[0]* (0x03+0xff) se trunca a 8 bits y se almacena en *z[0]*, sin afectar el resultado de *z[1]* que debe ser 0 en el ejemplo. Por esta razón no sirve calcular *z = x + y*, porque así *z[1]* sería incorrectamente 1.

Restricción: Ud. no puede usar los operadores de multiplicación, división o módulo (* / %). Use eficientemente los operadores de bits.

Ayuda: Está permitido calcular el resultado haciendo 8 sumas por medio de un ciclo de 8 iteraciones. Pero astutamente se puede hacer con solo 2 sumas, recurriendo a un uso inteligente de máscaras de bits.

Pregunta 2

Programe la función *palabras(str)* que transforma el string *str* dejando solo las palabras contenidas en *str* que estén formadas por caracteres alfabéticos. Además retorna el número de palabras encontradas. Las palabras quedarán en el mismo string *str* separadas por un espacio en blanco. Este es un ejemplo de uso:

```
char str[]=
 "  return   ( 'a'<=ch && ch<='z') || ('A'<=ch && ch<='Z');";
int n= palabras(str);
// n=9 y str es "return a ch ch z A ch ch Z";
```

Restricciones: No use el operador de subindicación de arreglos [ ] ni su equivalente *(p+i), use aritmética de punteros. No puede pedir memoria con *malloc* ni declarar arreglos.

Ayuda: Note que el string del ejemplo le servirá para determinar si un carácter es alfabético. Declare 2 punteros. Use uno para recorrer los caracteres del string y el otro para ir almacenando los caracteres que permanecen en el string.

Pregunta 3

Programe la función *cortar* definida como:

```
typedef struct nodo {
  char *pal;
  struct nodo *prox;
} Nodo;
void cortar(Nodo **plista, char *pal, Nodo **pres);
```

Esta función recibe en **plista* una lista simplemente enlazada en donde cada nodo almacena una palabra. Las palabras de la lista están ordenadas alfabéticamente. El segundo parámetro *pal* es una palabra que señala un punto de corte para la lista enlazada. Ud. debe cortar la lista dejando en **plista* la porción de la lista que antecede alfabéticamente a *pal* y en **pres* el resto de la lista. En el siguiente ejemplo de uso, la lista *h* ha sido creada con 5 nodos que almacenan las palabras "a", "b", "c", "d" y "e". Luego se invoca *cortar* como se indica en este código:

```
Nodo *h= …;
Nodo *r;
cortar(&h, "c", &r);
```

La siguiente figura muestra el cambio que produce la invocación de cortar en los punteros *h*, *r* y la lista enlazada.

Restricción: No puede usar *malloc*. Debe reutilizar los nodos que recibe en **plista*.