```
template <class T> class dll_node_t {
public:
  dll_node_t();
  dll_node_t(const T&);
  ~dll_node_t(void);
  T get_data(void) const;
  dll_node_t<T>* get_next(void) const;
  dll_node_t<T>* get_prev(void) const;
  void set_data(const T&);
  void set_next(dll_node_t<T>*);
  void set_prev(dll_node_t<T>*);
  ostream& write(ostream& = cout) const;

private:
  dll_node_t<T>* prev_;
  T              data_;
  dll_node_t<T>* next_;
};
```

```
template <class T> class dll_t {
public:
  dll_t(void);
  ~dll_t(void);
  dll_node_t<T>* get_tail(void);
  dll_node_t<T>* get_head(void);
  int get_size(void);
  bool empty(void);
  void insert_tail(dll_node_t<T>*);
  void insert_head(dll_node_t<T>*);
  dll_node_t<T>* extract_tail(void);
  dll_node_t<T>* extract_head(void);
  void remove(dll_node_t<T>*);
  ostream& write(ostream& = cout);

private:
  dll_node_t<T>* head_;
  dll_node_t<T>* tail_;
  int            sz_;
};
```

```
class sparse_vector_t {
 public:
  sparse_vector_t(const int = 0);
  sparse_vector_t(const vector_t<double>&,
                  const double = EPS);
  sparse_vector_t(const sparse_vector_t&);
  sparse_vector_t& operator=(const sparse_vector_t&);
  ~sparse_vector_t();
  int get_nz(void) const;
  int get_n(void) const;
  pair_double_t& at(const int);
  pair_double_t& operator[](const int);
  const pair_double_t& at(const int) const;
  const pair_double_t& operator[](const int) const;
  void write(std::ostream& = std::cout) const;

 private:
  pair_vector_t pv_;
  int nz_;
  int n_;
};
```

```
template<class T> class pair_t {
public:
  pair_t(void);
  pair_t(T, int);
  ~pair_t(void);
  T get_val(void) const;
  int get_inx(void) const;
  void set(T, int);
  istream& read(istream& is = cin);
  ostream& write(ostream& os = cout) const;

private:
  T val_;
  int inx_;
};
```

EJERCICIO 1. Implementar el operador * que devuelve el producto escalar de dos vectores dispersos (*sparse_vector_t*), con la siguiente cabecera:

```
double operator*(const sparse_vector_t& a, const sparse_vector_t& b)
```

Solución:

```
double operator*(const sparse_vector_t& a, const sparse_vector_t& b)
{
  assert(a.get_n() == b.get_n());

  const int a_sz = a.get_nz();
  const int b_sz = b.get_nz();

  double p = 0;
  int i = 0, j = 0;
  while (i < a_sz && j < b_sz)
  { const int a_inx = a[i].get_inx();
    const int b_inx = b[j].get_inx();

    if (a_inx == b_inx)     p += a[i++].get_val() * b[j++].get_val();
    else if (a_inx < b_inx) i++;
    else                    j++;
  }
  return p;
}
```

?

EJERCICIO 2. Dado un objeto *pw* de la clase *vector_t<char>* de longitud *n* que almacenará diferentes contraseñas (passwords), y el siguiente alfabeto:

const char ALPHABET[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
const int ALPHABET_size = (int)sizeof(ALPHABET) - 1;

se pide desarrollar un procedimiento **recursivo** en C++ con la siguiente cabecera:

```
void hacking(vector<char>& pw, const int i = 0)
```

que genere por pantalla todas las cadenas de caracteres de longitud *n* basadas en el alfabeto anterior. Por ejemplo, si ejecutamos *hacking(pw)* teniendo *pw* un tamaño *n* igual a 4, el resultado debería ser:

AAAA AAAB AAAC AAAD AAAE AAAF AAAG AAAH AAAI AAAJ AAAK AAAL AAAM AAAN AAAO AAAP AAAQ AAAR AAAS AAAT AAAU AAAV AAAW AAAX AAAY AAAZ AAA0 AAA1 AAA2 AAA3 AAA4 AAA5 AAA6 AAA7 AAA8 AAA9 AABA AABB AABC AABD AABE AABF AABG AABH AABI AABJ AABK AABL AABM AABN AABO AABP AABQ AABR AABS AABT AABU AABV AABW AABX 9993 9994 9995 9996 9997 9998 9999

NOTA: El último parámetro es el índice que itera sobre la cadena de caracteres que, por defecto, empieza en 0. En la evaluación de este ejercicio se tendrá muy en cuenta la implementación del caso base y del caso general, y la eficiencia de los bucles (si fueran necesarios).

Solución:

```
void hacking(vector<char>& pw, const int i = 0)
```

```
{ if (i == pw.size())
    cout << pw << " "; //endl;
  else
    for (int j = 0; j < ALPHABET_size; j++)
    { pw[i] = ALPHABET[j];
      hacking(pw, i + 1);
    }
}
```

EJERCICIO 3. Para la clase *matrix_t<int>*, desarrollar el método especializado

```
template<> bool matrix_t<int>::is_estocastica(void);
```

que determina si una matriz cuadrada, es estocástica o no, es decir, si no tiene valores negativos y la suma de cada columna es 1.

Solución:

```
bool is_estocastica(void)
{ int estocastica = 1;
  int suma;
  int i,j;

  //Comprobamos elementos negativos
  for (i = 0;i < TAM; i++)
  { for (j = 0;j < TAM; j++)
    { if (M[i][j] < 0)
        estocastica = 0; //return 0;
    }
  }

 //Si cumple el primer requisito, Comprobamos el de la suma de las columnas
 if (estocastica)
 for (i = 0; i < TAM; i++)
 { suma = 0;
 for(j = 0; j < TAM; j++)
 suma = suma + M[j][i];
 if (suma ! = 1)
 estocastica=0; //return 0;

 }
 return estocastica; //return 1;
 }
```

EJERCICIO 4. Desarrollar un algoritmo recursivo que retorne *true* si dada una frase dentro de un *vector_t<char>* es palíndroma o no, y con la siguiente cabecera:

```
bool is_palindrome(const vector_t<char>& s, const int i, const int d);
```

Un palíndromo es una palabra o frase que se lee igual de izquierda a derecha y de derecha a izquierda, sin tener en cuenta los espacios ni los caracteres especiales. Por ejemplo, el resultado esperado sería true para la siguiente cadena de entrada: ['Y','O','H','A','G','O','Y','O','G','A','H','O','Y']

Solución:

```
bool is_palindrome(const vector_t<char>& s, const int i, const int d)
{ if (i >= d) return true;
  return (s[i] == s[d] && is_palindrome(s, i + 1, d - 1));
}
```

EJERCICIO 5. Desarrollar el siguiente procedimiento

```
void merge(dll_t<int>& L1, dll_t<int>& L2, dll_t<int>& R)
```

que fusiona dos listas de enteros *L1* y *L2*, y devuelve el resultado en *R*, estando las tres listas ordenadas de menor a mayor.

Solución:

```
void merge(dll_t<int>& L1, dll_t<int>& L2, dll_t<int>& R)
{ dll_node_t<int> *ptr1 = L1.get_head(), *ptr2 = L2.get_head();

  while (ptr1 != NULL && ptr2 != NULL) {
    if (ptr1->get_data() <= ptr2->get_data()) {
      //cout << ptr1->get_data() << " ";
      R.insert_tail(new dll_node_t<int>(ptr1->get_data()));
      ptr1 = ptr1->get_next();
    }
    else if (ptr1->get_data() > ptr2->get_data()) {
      //cout << ptr2->get_data() << " ";
      R.insert_tail(new dll_node_t<int>(ptr2->get_data()));
      ptr2 = ptr2->get_next();
    }
  }

  while (ptr1 != NULL) {
    R.insert_tail(new dll_node_t<int>(ptr1->get_data()));
    ptr1 = ptr1->get_next();
  }

  while (ptr2 != NULL) {
    R.insert_tail(new dll_node_t<int>(ptr2->get_data()));
    ptr2 = ptr2->get_next();
  }
}
```

Última modificación: miércoles, 12 de julio de 2023, 10:19