TEMA 4

Vectores

Emma Rollón
erollon@cs.upc.edu

Departamento de Ciencias de la Computación

Índice

- Introducción
- Sintaxis
  - Declaración
  - Uso
  - Paso de parámetros
    - por referencia
    - por valor (simulado)
  - Retorno de función
- Esquemas algorítmicos
  - Recorrido
  - Búsqueda
    - vector no ordenado
    - vector ordenado
- String (vector de char)
- Typedef
- Ordenación (biblioteca algorithm)

Introducción

En matemáticas

$$v = (e_1, e_2, \ldots, e_n)$$

- Secuencia de escalares
- El tamaño es fijo
- Acceso directo por índice
- Índices válidos: $1, \ldots, n$

En informática

	0	1	...	n - 1
v:	e_0	e_1	$\ldots$	e_{n-1}

- Secuencia de tipo_de_datos
- El tamaño se puede variar
- Acceso directo por índice
- Índices válidos: $0, \ldots, n-1$

- Un vector es un tipo de datos que representa/agrupa una secuencia de valores bajo el mismo nombre.
- Un vector ocupa memoria proporcional a su tamaño.
- Sólo podremos acceder a aquellos índices que sean válidos.

Declaración de un vector

Sintaxis:

```
#include <vector>          // Biblioteca necesaria
```

```
vector<T> nombre_var(S, I);  // I es de tipo T
```

- T: tipos de datos
- S: tamaño del vector (por defecto es 0)
- I: valor inicial de los elementos del vector (si no se indica será inválido)

Los parámetros S, I son opcionales:

- Si sólo indicas uno, será S.
- Si indicas dos, serán S, I.

Ejemplos:

```
int main() {
    vector <int> v;         // vector de enteros llamado v que tiene 0 elementos
    vector <int> w(2);      // vector de enteros de 2 elementos, cada elemento tiene valor
                            // inválido
    vector <int> z(2, 4);   // vector de enteros de 2 elementos, cada elemento con valor 4
}
```

Uso de vectores

Acceso: siempre por índice

```
nombre_var[E]
```

- E: expresión entera
- Se accede a la posición E del vector nombre_var (sirve tanto para consultar como para actualizar)

```
int main() {
    vector <int> v(3, 1);
    int x = v[0];       // consulto valor posición 0 de v
    v[0] = 10;          // actualizo su valor
    cout << x << " " << v[0] << endl;
}
```

Alerta!

Al ejecutar el programa, si se accede a una posición no válida de un vector, el programa se para y te da el error:

```
container with out-of-bounds index
```

Es un error de ejecución.

Uso de vectores

```
// Pre: n >= 0, entero
// Post: ??
int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; ++i) v[i] = i*i;
    for (int i = 0; i < n; ++i) cout << v[i] << endl;
}
```

```
// Pre: ??
// Post: ??
int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    v[0] = 1;
    for (int i = 1; i < n; ++i) v[i] = 2*v[i-1];
    for (int i = 0; i < n; ++i) cout << v[i] << endl;
}
```

```
// Pre: ??
// Post: ??
int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    v[0] = v[1] = 1;
    for (int i = 2; i < n; ++i) v[i] = v[i-1] + v[i-2];
    for (int i = 0; i < n; ++i) cout << v[i] << endl;
}
```

Uso de vectores

Asignación (copia) de un vector a otro:

```
int main() {
    // Opción 1: copia de un vector que se ha modificado a lo
    //           largo del código
    vector<int> v(3, 1);
    (...)
    vector <int> w = v;

    // Opción 2: copia de un vector que se crea sin nombre
    vector<int> z;
    z = vector<int> (20, 5);

    // Opción 3: mala idea
    vector <int> vect(20, 5);
    vector <int> copia(10, 8);
    copia = vect;
}
```

La copia de vectores es una operación costosa.

Uso de vectores

Saber número de elementos:

```
int n = nombre_var.size();
```

- La función .size() retorna el número de elementos del vector sobre el que se llame.

Lectura/escritura de vectores:

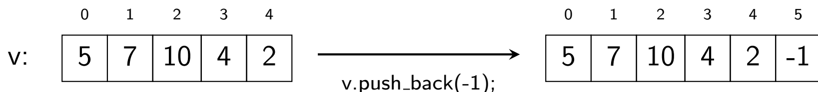
```
vector <T> v;
cin >> v;     // ERROR de compilación: cin sólo sobre
              //       tipos de datos simples
cout << v;    // ERROR de compilación: cout sólo sobre
              //       tipos de datos simples
```

Uso de vectores

Añadir un nuevo elemento:

```
nombre_vector.push_back(valor_T);
```

- Al vector *nombre_vector* se le añade un nuevo último elemento con valor *valor_T* y tipo de datos T (el mismo que el del vector).

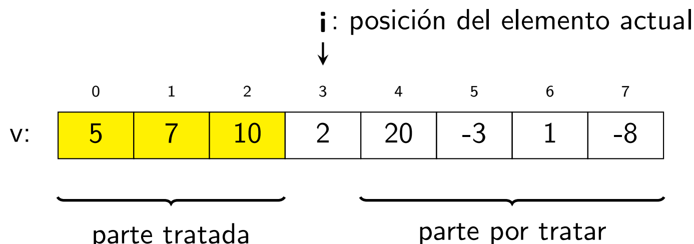


Borrar último elemento del vector:

```
nombre_vector.pop_back();
```

- Al vector *nombre_vector* se le quita su último elemento.

Fíjate que:



Un vector es una secuencia almacenada en memoria:

- A priori sé su número de elementos: v.size()
- Al elemento actual lo accederemos por su posición i.
- La parte tratada es $v[0, ..., i-1]$.
- El vector $v[a, ..., b]$ con $a > b$ representa un vector vacío.

Todo el razonamiento explicado para secuencias es aplicable.

Un primer ejemplo: cuántos iguales al último

Escribe un programa tal que dado un entero $n \geq 0$ y una secuencia de n enteros, escriba cuántas veces el último elemento de la secuencia está repetido.

E: 12
3 5 1 1 10 6 8 1 2 4 6 1

S: 4

Fíjate que:

- tenemos que leer el último elemento para poder contar iguales
- los valores de la entrada se leen/consumen en orden secuencial
- por tanto, hay que almacenarlos en algún "sitio" $\rightarrow$ en un vector

¡Alerta!

Almacenar una secuencia de entrada en un vector si no es necesario es MUY ineficiente.

Un primer ejemplo: cuántos iguales al último

```
// Input: número de elementos n >= 0;
//        le sigue una secuencia de enteros de n elementos
// Output: cuántos elementos son iguales al último
//        (éste incluído)
int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; ++i) cin >> v[i];

    int count = 0;
    for (int i = 0; i < n; ++i) {
        if (v[i] == v[n - 1]) ++count;
    }
    cout << count << endl;
}
```

Paso de parámetros

Por referencia:

- Todo lo que se ha explicado para los tipos simples también se aplica aquí
- Nada nuevo

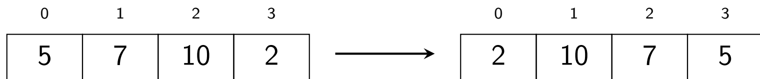
Por valor:

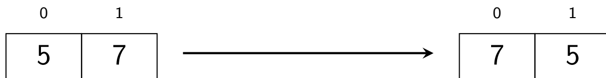
- Todo lo que se ha explicado para los tipos simples también se aplica aquí
- Como es así, este tipo de paso de parámetro es costoso en tiempo y espacio
- Utilizaremos un paso por valor simulado: paso por referencia constante

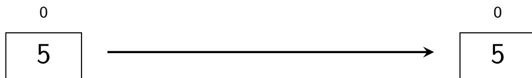
Paso de parámetros: paso por referencia

Escribe un prodecimiento tal que dado un vector de enteros válido, lo invierta.

<u>Ejemplos:</u>

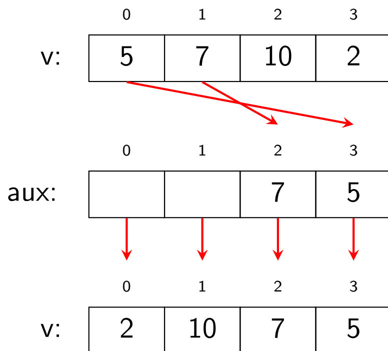






Paso de parámetros: paso por referencia

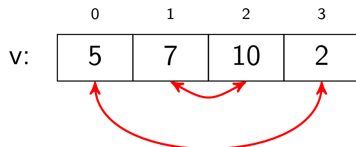
Algoritmo 1:




```
void invertir(vector<int>& v) {
    int n = v.size();
    vector<int> aux(n);
    for (int i = 0; i < n; ++i) aux[n - 1 - i] = v[i];
    for (int i = 0; i < n; ++i) v[i] = aux[i];
}
```

Paso de parámetros: paso por referencia

Algoritmo 2:




```
void invertir(vector<int>& v) {
    int n = v.size();
    for (int i = 0; i < n/2; ++i) {
        int aux = v[i];
        v[i] = v[n - 1 - i];
        v[n - 1 - i] = aux;
    }
}
```

Cuestiones:

- Es necesario tratar el elemento de la posición n/2?
- Qué pasaría si la condición fuera $i < n$?

Paso de parámetros: paso por valor

```
void nombre_tarea(vector<T> v, ...) {
    /* Al hacer la llamada, antes de ejecutar el código se
       hará la asignación:
           v = vector con el que se realiza la llamada;
        Esta copia es costasa tanto en tiempo como en espacio
    */
    ...
    v[p] = E; // el vector v se modifica, el de la llamada no
    ...
}

int main() {
    int n;
    cin >> n;
    vector<T> w(n);
    for (int i = 0; i < n; ++i) cin >> w[i];
    // llamada al procedimiento
    nombre_tarea(w, ...);
    (...)
}
```

Paso de parámetros: paso por valor

Para evitar la copia de los vectores $\Longrightarrow$ paso por referencia constante.

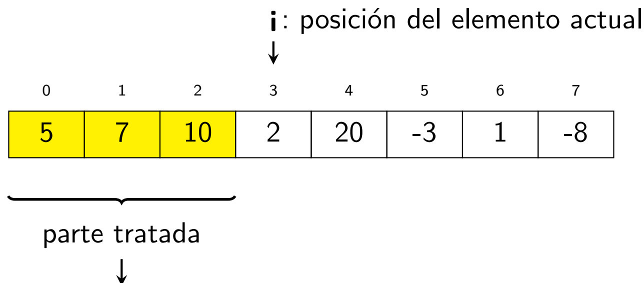
```
void nombre_tarea(const vector<T>& v, ...) {
    /* v y el vector con el que se hace la llamada son el
       mismo (es por referencia) --> no se hace copia
      Al poner const se prohíbe modificar el contenido de v
    */
    ...
    v[p] = E;        // ERROR de compilación
    ...
}

int main() {
    (...)
    // llamada al procedimiento
    nombre_tarea(w, ...);
    (...)
}
```

Paso de parámetros: paso por valor

Escribe una función tal que dado un vector de enteros válido, retorne la media de sus elementos.

Algoritmo:



Recuerda que el número de elementos de un vector v te lo da v.size()

Paso de parámetros: paso por valor

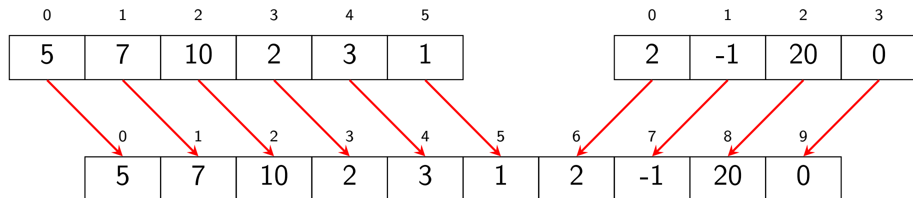
Implementación:

```
// Pre: v es un vector de enteros válido
// Post: retorna la media de sus elementos
double average(const vector<int>& v) {
    int suma = 0;
    int n = v.size();
    for (int i = 0; i < n; ++i) suma = suma + v[i];
    return suma/double(n);    // Recuerda que / es división
                              // real o entera dependiendo
                              // de los operandos
}
```

Retorno de una función

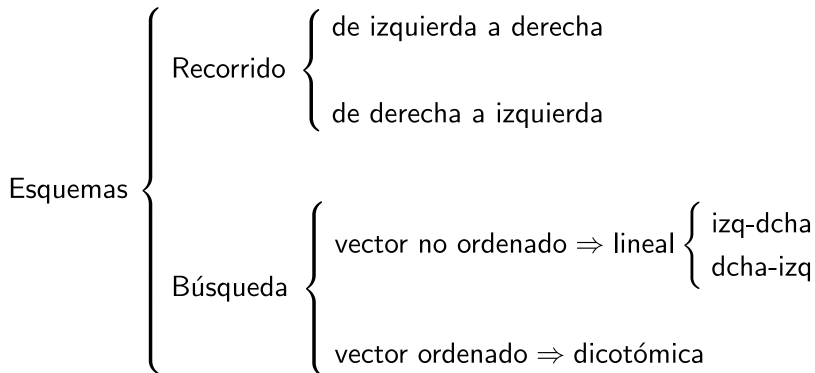
Escribir una función tal que dados dos vectores v1 y v2, retorne un vector con su concatenación.

Ejemplo:




```
vector<int> concatenar(const vector<int>& v1, const vector<int>& v2) {
    int n1 = v1.size();
    int n2 = v2.size();
    vector<int> res(n1 + n2);
    for (int i = 0; i < n1; ++i) res[i] = v1[i];
    for (int i = 0; i < n2; ++i) res[n1 + i] = v2[i];
    return res;
}
```

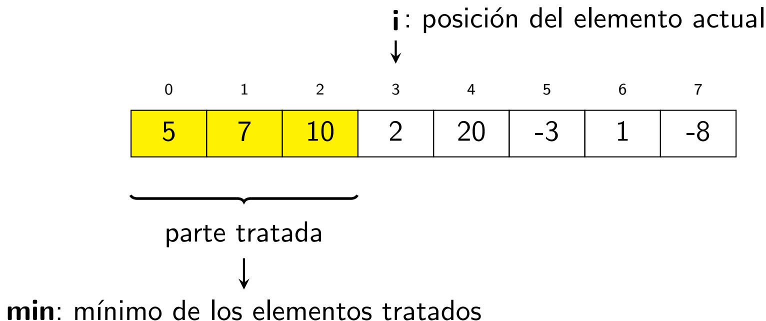
Esquemas algorítmicos



Esquemas algorítmicos: Recorrido

Escribir una función tal que dado un vector no vacío de enteros retorne su valor mínimo.

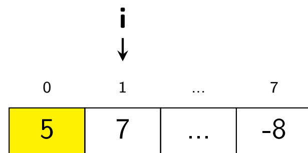
Algoritmo:



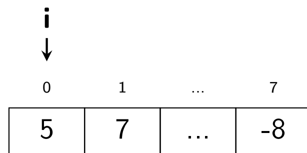
Esquemas algorítmicos: Recorrido

Cuál será la inicialización?

Opción 1:



Opción 2:



- Como la Pre nos dice que el vector es no vacío $\Longrightarrow$ opción 1.
- Si la Pre no nos dijera nada sobre su longitud (puede ser vacío) $\Longrightarrow$ opción 2.

Implementación:

```
// Pre: v es un vector no vacío de enteros válido
// Post: retorna su valor mínimo
int minimo_vector(const vector<int>& v) {
    int min = v[0];
    for (int i = 1; i < v.size(); ++i) {
        if (min > v[i]) min = v[i];
    }
    return min;
}
```

Otros ejemplos: (transparencias del material docente)

1. Picos de un vector: dado un vector de enteros, contar el número de picos que contiene. Un pico es el último elemento de una secuencia ascendente o descendente de valores.

2. Normalizar una secuencia: dada una secuencia de naturales, en donde primero se indica el número de elementos que tendrá la secuencia a tratar, escribe esa misma secuencia normalizada (es decir, restando el elemento más pequeño a todos los elementos de la secuencia).

3. Clasificación de elementos: dado un vector de enteros y dos enteros x, y que representan el intervalo $[x, y]$ ordenar los elementos del vector de tal manera que los elementos más pequeños que x en la parte izquierda del vector, los más grandes que y en la parte derecha, y los que pertenecen al intervalo en la parte central del vector. El orden relativo de los elementos es indiferente.

Esquemas algorítmicos: Búsqueda lineal

El vector no está ordenado $\Longrightarrow$ búsqueda lineal:

- Empiezo desde un extremo del vector (primer o último elemento).
- Avanzo al siguiente elemento en orden según índice (hacia izquierda o hacia derecha).
- Hasta que encuentro lo que busco o llego al otro extremo del vector.

```
// Pre: --
// Post: retorna la primera posición de x en v si existe, -1 en caso contrario
int primera_pos_valor(const vector<int>& v, int x) {
    for (int i = 0; i < v.size(); ++i) {
        if (v[i] == x) return i;
    }
    return -1;
}
```

```
// Pre: --
// Post: retorna la última posición de x en v si existe, -1 en caso contrario
int ultima_pos_valor(const vector<int>& v, int x) {
    int n = v.size();
    for (int i = n - 1; i >= 0; --i) {
        if (v[i] == x) return i;
    }
    return -1;
}
```

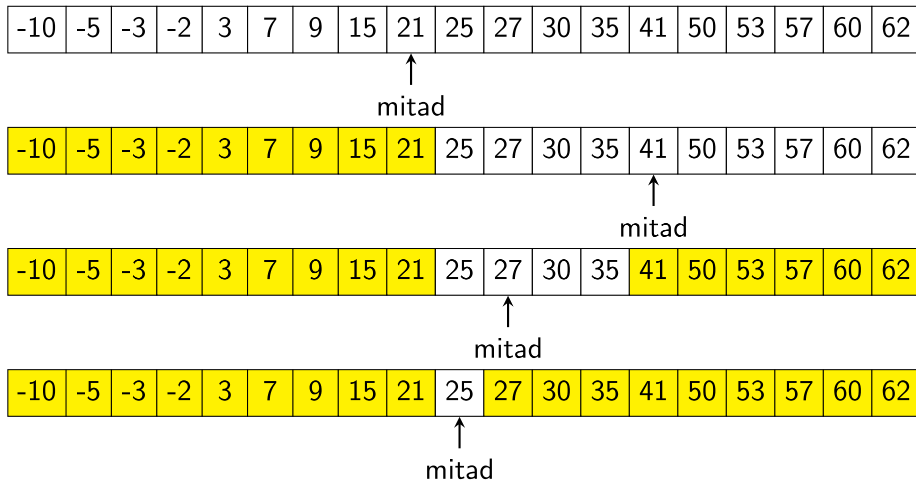
Esquemas algorítmicos: Búsqueda dicotómica

El vector está ordenado $\Longrightarrow$ búsqueda dicotómica:

- Miro el valor que hay a la mitad del vector
- Si ese valor es mayor que el buscado, los elementos a su derecha todavía lo son más, así es que el que busco no estará en esas posiciones.
- Si ese valor es menor que el buscado, los elementos a su izquierda todavía lo son más, así es que el que busco no estará en esas posiciones.
- Repetir este razonamiento hasta que encuentro lo que busco o descarto todos los elementos.

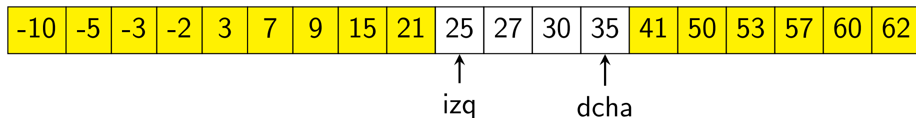
Esquemas algorítmicos: Búsqueda dicotómica

Escribir una función tal que dado un vector v de enteros ordenado y un entero x, retorne la posición de x en v (si existe), y -1 en caso contrario.



Esquemas algorítmicos: Búsqueda dicotómica

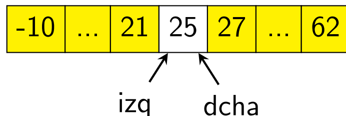
1. ¿Cómo me puedo acordar de la zona que no es amarilla?



2. ¿Cuál es el significado de esas variables? (invariante)

   ⇒ x todavía puede estar en $v[izq, ..., dcha]$

3. ¿Cómo podemos detectar que el elemento no existe en el vector?



x = 26 ?

x = 24 ?

Esquemas algorítmicos: Búsqueda dicotómica

```
int busqueda_binaria(const vector<int>& v, int x) {                    // ITERATIVA
    int izq = 0;
    int dcha = v.size() - 1;
    while (izq <= dcha) {
        int m = (dcha + izq)/2;
        if (v[m] < x) izq = m + 1;
        else if (x < v[m]) dcha = m - 1;
        else return m;
    }
    return -1;
}
```

```
int busqueda_binaria(const vector<int>& v, int x, int izq, int dcha) {   // RECURSIVA
    if (izq > dcha) return -1;
    int m = (dcha + izq)/2;
    if (v[m] < x) return busqueda_binaria(v, x, m + 1, dcha);
    else if (x < v[m]) return busqueda_binaria(v, x, izq, m - 1);
    else return m;
}

int main() {
    int n, x;
    cin >> x >> n;
    vector<int> v(n);
    ...
    int pos = busqueda_binaria(v, x, 0, v.size() - 1);
    ...
}
```

String

El tipo de datos string (que hasta ahora lo hemos tratado como un tipo de datos simple), es en realidad un vector de char $\Longrightarrow$ Todo lo explicado para vectores sirve para los strings.

Por ejemplo, el string:

"clase"

lo podemos tratar como el vector de chars:

0	1	2	3	4
'c'	'l'	'a'	's'	'e'

Eso quiere decir que podemos:

- saber su número de carácteres
- acceder a una posición para consultar o modificar su valor

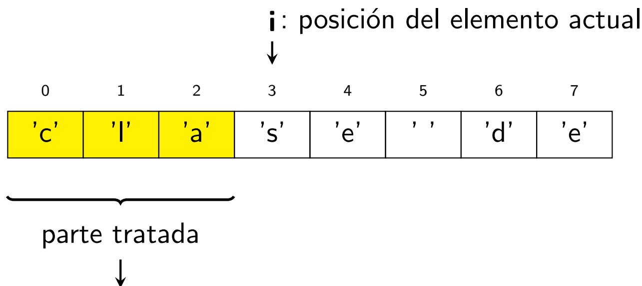
Cuidado con la declaración:

```
string s(10);          // ERROR de compilación        /* Recordad que:
string s(10, 'a');     // ok                              #include <string>
string s = "un string cualquiera";  // ok              */
```

String: Ejemplo 1

Escribir una función tal que dado un string retorne su número de vocales.

Algoritmo:



String: Ejemplo 1

Implementación:

```
// Pre: --
// Post: retorna true si c es una vocal
bool es_vocal(char c) {
    if (c == 'a' or c == 'e' or c == 'i' or c == 'o' or c == 'u')
        return true;
    if (c == 'A' or c == 'E' or c == 'I' or c == 'O' or c == 'U')
        return true;
    return false;
}

// Pre: --
// Post: retorna el número de vocales del string s
int num_vocales(const string& s) {
    int n_voc = 0;
    for (int i = 0; i < s.size(); ++i) {
        if (es_vocal(s[i])) ++n_voc;
    }
    return n_voc;
}
```

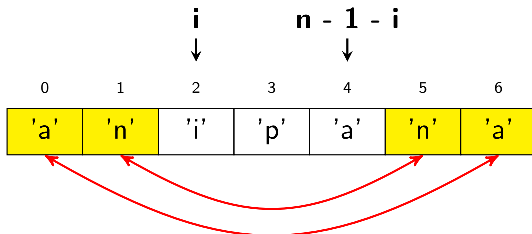
String: Ejemplo 2

Escribe una función tal que dada una palabra retorne true si es palíndroma, false en caso contrario. Un palíndromo es un string que se lee igual de izquierda a derecha que de derecha a izquierda.

Ejemplos:

"reconocer" $\longrightarrow$ true
"anipana" $\longrightarrow$ false

Algoritmo:



String: Ejemplo 2

Implementación:

```
// Pre: s es una palabra
// Post: retorna true si s es palíndroma,
//       false en caso contrario
bool palindromo(const string& s) {
    int n = s.size();
    for (int i = 0; i < n/2; ++i) {
        if (s[i] != s[n - 1 - i]) return false;
    }
    return true;
}
```

String: Ejemplo 3

Escribir una función tal que dados dos strings subs y s, retorne la posición más pequeña de s a partir de la cual subs es un substring de s, -1 en caso de que subs no sea subtring de s.

Ejemplos:

subs = “theor”; s = “the big bang theory” $\longrightarrow$ 13

subs = “bing”; s = “the big bang theory” $\longrightarrow$ -1

Algoritmo:

- Para cada posición válida i de s:
- Buscaré si desde esa posición i de s está el substring subs
- Si lo encuentro entonces retorno i
- Retorno -1 (para ninguna posición he encontrado que fuera substring)

String: Ejemplo 3

Implementación:

```
// Pre: 0 <= i < s.size()
// Post: retorna true si subs es un substring de s desde s[i]
bool is_substring(const string& subs, const string& s, int i) {
    int j = 0;
    while (i < s.size() and j < subs.size() and s[i] == subs[j]) {
        ++i;
        ++j;
    }
    return j == subs.size();
}

// Pre: --
// Post: retorna la posición de s a partir de la cual subs es un substring,
//       -1 en caso de que subs no sea substring de s.
int substring(const string& subs, const string& s) {
    for (int i = 0; i < s.size(); ++i) {
        if (is_substring(subs, s, i)) return i;
    }
    return -1;
}
```

Fíjate que la condición del for de substring podría ser:

$$i < s.size() - subs.size()$$

Dando significado al índice del vector

Dados dos frases en letras minúsculas acabadas en '.', decir si son anagramas o no. Dos frases son anagramas si ambas tienen las mismas letras el mismo número de veces.

Algoritmo:

- Leer la primera secuencia contando cuántas veces aparece cada una de sus letras. Dónde lo almaceno?
- Leer la segunda secuencia descontando el número de veces que aparece cada letra. Si en algún punto algún contaje se hace negativo, la frase no es anagrama.
- Comprobar si todos los contadores son 0. Si lo son, es anagrama. Si no, no lo es.

'a' ↕ 0	'b' ↕ 1	'c' ↕ 2	'd' ↕ 3	'e' ↕ 4	'f' ↕ 5	... ...	'z' ↕ 26
5	7	1	2	0	3	...	0

Dando significado al índice del vector

Implementación:

```
int main() {
    const int ALFABETO = 'z' - 'a' + 1;    // Constante: su valor inicial no cambia
    vector<int> cont(ALFABETO, 0);
    // tratar primer secuencia
    char c;
    cin >> c;
    while (c != '.') {
        ++cont[c - 'a'];
        cin >> c;
    }
    // tratar segunda secuencia
    bool correcto = true;
    cin >> c;
    while (correcto and c != '.') {
        --cont[c - 'a'];
        if (cont[c - 'a'] < 0) correcto = false;
        cin >> c;
    }
    // comprobar contadores
    int i = 0;
    while (correcto and i < ALFABETO) {
        if (cont[i] != 0) correcto = false;
        ++i;
    }
    // escribir resultado
    if (correcto) cout << "anagrama" << endl;
    else cout << "NO anagrama" << endl;
}
```

Dando significado al vector: polinomios

Un vector puede representar un polinomio. Hay dos posibles formas de representación:

	0	1	2	3
p:	5	7	10	4

$\longrightarrow p = 5 + 7x + 10x^2 + 4x^3$

	0	1	2	3
p:	5	7	10	4

$\longrightarrow p = 5x^3 + 7x^2 + 10x + 4$

Ejercicios:

1. Evaluación de un polinomio: con y sin regla de Horner
2. Producto de dos polinomios

Typedef

Sintaxis:

```
typedef tipo_de_datos_existente nuevo_nombre;

// otra sintaxis: compilar con -std=c++11 o superior
using nuevo_nombre = tipo_de_datos_existente;
```

Semántica:

- *nuevo_nombre* es sinónimo de *tipo_de_datos_existente*.

```
#include <vector>
#include <iostream>
using namespace std;

typedef vector<int> Polinomio;  // using Polinomio = vector<int>;

int evaluar_polinomio(const Polinomio& p, int x) {
    ...
}

int main() {
    int x, n;
    cin >> x >> n;
    Polinomio p(n);
    for (int i = 0; i < n; ++i) cin >> p[i];
    cout << evaluar_polinomio(p, x) << endl;
}
```

Aprovechando que el vector está ordenado

Cuando el vector está ordenado podemos realizar algunas tareas de forma más eficiente que si no lo estuviera (esto es, sin bucles anidados).

Ejemplos:

1. Segmento nulo más largo en un vector ordenado.
2. Número de elementos comunes de dos vectores ordenados.
3. Fusión de dos vectores ordenados.
4. Diferencia de dos vectores ordenados:
   - Los vectores tienen repetidos y la diferencia también tiene repetidos.
   - Los vectores tienen repetidos y en la diferencia no queremos repetidos.
5. Intersección de dos vectores ordenados:
   - Los vectores no tienen repetidos.
   - Los vectores tienen repetidos y en la intersección no queremos repetidos.

(Algunos en transparencias del material docente)

Segmento nulo más largo en un vector ordenado

Dado un vector de enteros ordenado, decir los índices del segmento nulo más largo. Un segmento nulo es un subconjunto de posiciones consecutivas tales que la suma de sus elementos es 0.

-20	-10	-8	-5	-4	-2	1	5	6	7	15

-20	-10	-8	-5	-4	-2	1	5	6	7	15

segmento nulo

-20	-10	-8	-5	-4	-2	1	5	6	7	15

segmento nulo más largo

Segmento nulo más largo en un vector ordenado

Idea:

- Partimos calculando la suma de todos los elementos del vector.
- Si esa suma es positiva:

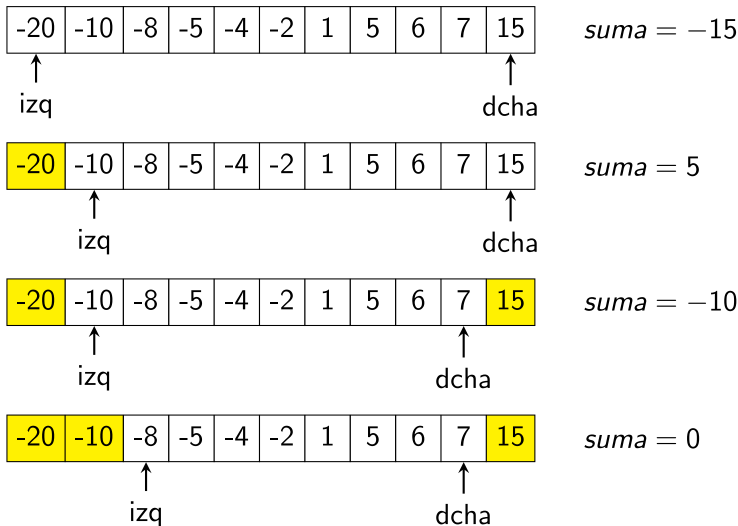
  $\Rightarrow$ el valor positivo más grande no forma parte del segmento nulo
  $\Rightarrow$ descartarlo
- Si esa suma es negativa:

  $\Rightarrow$ el valor negativo más pequeño no forma parte del segmento nulo
  $\Rightarrow$ descartarlo
- Repetir este razonamiento mientras la suma no esa 0:

  $\Rightarrow$ los que queden sin descartar son el segmento nulo más largo
  $\Rightarrow$ si no hay segmento nulo, se descartan todos, así es que la suma será 0

Segmento nulo más largo en un vector ordenado

Evolución:



Segmento nulo más largo en un vector ordenado

Implementación:

```
// Pre: v está ordenado ascendentemente
// Post: left, right son los índices del segmento nulo
//       más largo si existe; si no existe left > right
void segmento_nulo(const vector<int>& v, int& left, int& right) {
    int suma = suma_elementos(v);
    left = 0;
    right = v.size() - 1;
    // Inv: de v[left] a v[right] puede haber segmento nulo;
    //      suma es la suma de esos elementos
    while (suma != 0) {
        if (suma > 0) {
            suma = suma - v[right];
            --right;
        } else {
            suma =) suma - v[left];
            --left;
        }
    }
    // suma == 0
    // v[left, ...., right] es el segmento nulo
}
```

Ordenación (biblioteca algorithm)

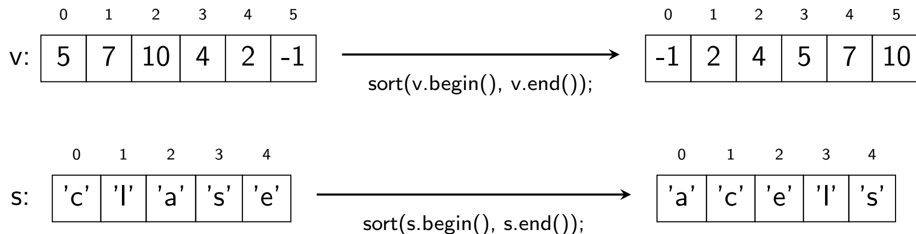
Sintaxis:

```
#include <algorithm>          // Biblioteca necesaria
```

```
sort(nombre_vector.begin(), nombre_vector.end());
```

Semántica: Se ordena todo el vector *nombre_vector*, en orden ascendente según el operador $<$ definido para el tipo de datos de ese vector.

Ejemplo:



Ordenación (biblioteca algorithm)

Sintaxis:

```
// Post: retorna true si a tiene que ir antes que b en el vector,
//       false en caso contrario
bool nom_func_bool(const T& a, const T& b) {...}
```

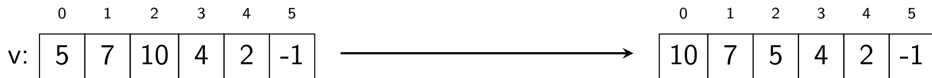
```
sort(nom_v.begin(), nom_v.end(), nom_func_bool);
```

Semántica:

Se ordena todo el vector *nombre_v*, en orden ascendente según la función booleana *nom_func_bool*. T es el tipo de datos del vector. El orden es estricto (es decir, *nom_func_bool(a, a)*$\rightarrow$ false).

Ejemplo:

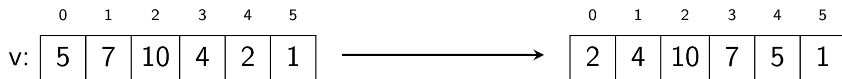
```
// quiero que a vaya antes que b en el vector cuando a > b
bool decreciente(int a, int b) {
    if (a > b) return true;
    else return false;
}
```



Ordenación (biblioteca algorithm)

Escribir un procedimiento tal que dado un vector de enteros, lo transforme de tal manera que sus elementos pares aparezcan antes que los elementos impares. Además, los elementos pares estarán ordenados de forma ascendente, mientras que los impares de forma descendente.

Ejemplo:




```
bool orden(int a, int b) {
    bool par_a = a%2 == 0;
    bool par_b = b%2 == 0;

    if (par_a != par_b) return par_a;      // tienen diferente paridad
    if (par_a) return a < b;               // los dos son pares
    return a > b;                          // los dos son impares
}

void separar(vector<int>& v) {
    sort(v.begin(), v.end(), orden);
}
```

¡Ordenar puede no ser buena idea!

Ordenar es una tarea costosa:

$\longrightarrow$ tiene un ¡doble bucle anidado!

Hay tareas que NO se tienen que resolver ordenando. Ejemplos:

- Encontrar el mínimo o el máximo: haciendo un recorrido del vector es suficiente

  $\longrightarrow$ un único bucle

- Contar cuántos elementos cumplen una condición como ser par, ser mayor que el máximo/mínimo/suma de los elementos de su izquierda, etc: haciendo un recorrido del vector es suficiente

  $\longrightarrow$ un único bucle

Siempre piensa que si lo puedes solucionar con un único bucle, entonces (en general) será mejor que ordenando el vector (a pesar de que visualmente utilices con sort menos líneas de código).