TEMA:

Tuplas

Emma Rollón
erollon@cs.upc.edu

Departamento de Ciencias de la Computación

Índice

- Introducción
- Sintaxis
  - Declaración
  - Uso
  - Paso de parámetros
    - por valor
      - estándar
      - simulado
    - por referencia
  - Retorno de función
- Tuplas anidadas
- Vectores de tuplas
  - Recorrido
  - Búsqueda
  - Ordenación
- Tuplas con vectores

Introducción

Una tupla es un tipo de datos que nos permite agrupar bajo un mismo nombre un conjunto fijo de valores con (posiblemente) diferentes tipos de datos.

$$\text{Reloj} \begin{cases} \text{hora (int)} \\ \text{min (int)} \\ \text{sec (int)} \end{cases} \qquad \text{Racional} \begin{cases} \text{num (int)} \\ \text{den (int)} \end{cases} \qquad \text{Film} \begin{cases} \text{title (string)} \\ \text{year (int)} \end{cases}$$

Cada valor de la tupla se denomina campo y tiene:

- un nombre (siempre en minúscula)
- un tipo de datos (cualquiera de los que hemos visto)

Declaración

Sintaxis:

```
...
using namespace std;

struct Nombre_tupla {
    tipo_de_datos campo1;
    ...;
    tipo_de_datos campoN;
};                          // <------- acaba con un ;

int main() {
    Nombre_tupla nombre_var;
    ...
}
```

Semántica:

Se crea el tipo de datos llamado Nombre_tupla. A partir de ese momento se pueden declarar variables de ese tipo de datos.

Observación

Nombre_tupla no tiene definido ninguno de los operadores de comparación:

$$>, >=, <, <=, ==, !=$$

Declaración (ejemplos)

```
struct Racional {
    int num;
    int den;       // den > 0
};

int main() {
    Racional r;
    ...
}
```

```
struct Reloj {
    int hora;     // 0 <= hora < 24
    int min;      // 0 <= min < 60
    int sec;      // 0 <= sec < 60
};

int main() {
    Reloj r;
    ...
}
```

```
struct Film {
    string title;
    int year;
};

int main() {
    Film peli;
    ...
}
```

Uso: acceso

Sintaxis:

```
nombre_var.nombre_campo
```

Semántica:

- Se accede al campo *nombre_campo* de la variable *nombre_var*.
- Para que sea correcto, el tipo de datos de la variable ha de tener un campo llamado así.
- Este acceso es válido tanto para consultar el valor de ese campo como para modificarlo.

Uso: acceso (ejemplos)

```
...
using namespace std;

struct Racional {
    int num;
    int den;      // den > 0
};

int main() {
    Racional r;
    cin >> r.num >> r.den;
    ....
    cout << r.num/double(r.den) << endl;
}
```

```
...
using namespace std;

struct Reloj {
    int hora;     // 0 <= hora < 24
    int min;      // 0 <= min < 60
    int sec;      // 0 <= sec < 60
};

int main() {
    Reloj r;
    cin >> r.hora >> r.min >> r.sec;
    ++r.sec;
    ...
    cout << r.hora << ':' << r.min << ...;
}
```

Uso: asignación

Sintaxis:

```
Tupla_A t;
// inicialización de los campos de t
...
// asignación a otra variable
Tupla_A copia = t;
```

Semántica:

Se hace una asignación campo a campo de la variable *t* a la variable *copia*.

```
int main() {
    Racional r;
    cin >> r.num >> r.den;
    Racional copia = r;          // lo que hace:   copia.num = r.num; copia.den = r.den;
    copia.den = copia.den + 5;
    cout << r.den << ' ' << copia.den << endl;
}
```

¡Alerta!

Depende del tipo de datos de los campos, esta asignación puede ser muy costosa! Si es así, pensar si es absolutamente necesario hacerla.

Paso de parámetros

Por valor:

- Recuerda que el paso por valor se hace sobre los parámetros de entrada
- En la llamada, se hace copia de esos valores.
- Por tanto, tenemos que pensar:
  - Paso por valor estándar: si la tupla tiene pocos valores (copia no costosa)

```
void mi_accion(Racional r) {
    ...
}
```

  - Referencia constante: si la tupla tiene muchos valores (copia costosa)

```
void mi_accion(const Persona& p) {    // tupla definida más adelante
    ...
}
```

Por referencia:

- Recuerda que el paso por referencia se hace sobre los parámetros de salida o de entrada/salida.
- Nada nuevo

Paso de parámetros: paso por valor estándar

Escribe una función tal que, dados dos racionales a y b (ambos con denominadores positivos), retorne cierto si a es mayor que b, false en caso contrario.

```
// Pre: a.den > 0, b.den > 0
// Post: retorna true si a mayor que b, false en caso contrario
bool mayor(Racional a, Racional b) {
    return a.num*b.den > b.num*a.den;
}
```

Escribe una función tal que, dados dos racionales a y b (ambos con denominadores positivos), retorne cierto si a es igual que b, false en caso contrario.

```
// Pre: a.den > 0, b.den > 0
// Post: retorna true si a igual que b, false en caso contrario
bool igual(Racional a, Racional b) {
    return a.num*b.den == b.num*a.den;
}
```

Paso de parámetros: paso por referencia

Escribe un procedimiento tal que, dado un reloj válido r, le sume un segundo.

```
// Pre: r es válido
// Post: suma un segundo al reloj y lo deja con el formato correcto
void suma_segundo(Reloj& r) {
    ++r.sec;
    if (r.sec == 60) {
        r.sec = 0;
        ++r.min;
        if (r.min == 60) {
            r.min = 0;
            ++r.hora;
            if (r.hora == 24) r.hora = 0;
        }
    }
}

// Pre: en la entrada hay 3 enteros que representan hora, minutos y segundos
// Post: escribir la hora, minutos y segundos de la entrada incrementada en 1 segundo
int main() {
    Reloj mi_r;     // mejor llamarlo r (objetivo: incidir en el paso por referencia)
    cin >> mi_r.hora >> mi_r.min >> mi_r.sec;
    suma_segundo(mi_r);
    cout << mi_r.hora << ' ' << mi_r.min << ' ' << mi_r.sec << endl;
}
```

Retorno de una función

Escribe una función tal que retorne un reloj inicializado a media noche.

```
// Pre: —
// Post: retorna un reloj inicializado a media noche
Reloj medianoche() {
    Reloj r;
    r.hora = 0;
    r.min = 0;
    r.sec = 0;
    return r;
}

// Uso de la función
int main() {
    ...
    Reloj mi_r = medianoche();
    ...
}
```

Tuplas anidadas

$$\text{Nif} \begin{cases} \text{dni (int)} \\ \text{letra (char)} \end{cases} \qquad \text{Persona} \begin{cases} \text{nombre (string)} \\ \text{nif (Nif)} \\ \text{edad (int)} \end{cases}$$

```
...
using namespace std;

struct Nif {
    int dni;          // dni > 0
    char letra;
};

struct Persona {
    string nombre;
    Nif nif;
    int edad;         // edad > 0
};

int main() {
    Persona p;
    p.nombre = "Arnau";   // cin >> p.nombre;
    p.edad = 14;
    p.nif.dni = 45789;      // p.nif es de tipo Nif (tiene dos campos: dni, letra)
    p.nif.letra = 'E';
    ...
    p.edad = p.edad + 6;
    cout << p.nif.dni << " " << p.edad;
}
```

Vectores de tuplas (recorrido)

Escribe una función tal que, dado un vector no vacío de racionales, retorne el racional mayor.

```
// Pre: v es no vacío
// Post: retorna el racional máximo de v
Racional max(const vector<Racional>& v) {
    int n = v.size();
    Racional m = v[0];
    for (int i = 1; i < n; ++i) {
        if (mayor(v[i], m)) m = v[i];
    }
    return m;
}
```

Vectores de tuplas (búsqueda)

Escribe una función tal que, dado un vector de racionales v y un racional r, retorne true si existe en v algún racional que represente el mismo valor que r, false en caso contrario.

```
// Pre: v es válido
// Post: retorna true si r está en v, false en caso contrario
bool esta(const vector<Racional>& v, Racional r) {
    int n = v.size();
    for (int i = 0; i < n; ++i) {
        if (igual(v[i], r)) return true;
    }
    return false;
}
```

Vectores de tuplas (ordenación)

Recuerda que sort (librería algorithm) ordena vectores:

- si el operador $<$ está definido sobre el tipo de datos del vector:

```
int main() {
    ...
    sort(v.begin(), v.end());
    ...
}
```

- si el operador $<$ no está definido sobre el tipo de datos del vector o queremos otro orden:

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

int main() {
    vector<T> v(...);
    ...
    sort(v.begin(), v.end(), nombre_funcion);
    ....
    }
```

¡Para que sea correcto, la llamada a nombre_funcion con el mismo valor para los dos parámetros tiene que retornar false!

Vectores de tuplas (ordenación)

Queremos ordenar decrecientemente un vector de racionales.

```
// Pre: a.den > 0, b.den > 0
// Post: retorna true si a mayor que b, false en caso contrario
bool mayor(Racional a, Racional b) {
    return a.num*b.den > b.num*a.den;
}

int main() {
    int n;
    cin >> n;
    vector<Racional> v(n);
    ...
    sort(v.begin(), v.end(), mayor);
    ...
}
```

¿Qué pasará con los racionales que representen el mismo valor?

- Ordenación estable: se mantiene el orden relativo en el vector v
- Ordenación no estable: no tiene por qué mantener el orden relativo en el vector v (sort es no estable)

Vectores de tuplas (ordenación)

Queremos ordenar un vector de films por los siguientes criterios:

(1) de más antigua a más reciente según año de producción;

(2) para las películas con el mismo año de producción, lexicográficamente creciente según título.

```
// Pre: —
// Post: retorna true si a cumple los criterios de ordenación sobre b,
//               false en caso contrario
bool comp(Film a, Film b) {
    if (a.year != b.year) return a.year < b.year;
    return a.title < b.title;
}

int main() {
    int n;
    cin >> n;
    vector<Film> pelis(n);
    ...
    sort(pelis.begin(), pelis.end(), comp);
    ...
}
```

Vectores de tuplas (ordenación)

Queremos ordenar un vector de personas por los siguientes criterios:

(1) lexicográficamente creciente por nombre;

(2) para las personas con un mismo nombre, decreciente respecto la edad;

(3) para las que también tengan la misma edad, creciente según el dni.

```
// Pre: —
// Post: retorna true si a cumple los criterios de ordenación sobre b,
//               false en caso contrario
bool comp(const Persona& a, const Persona& b) {
    if (a.nombre != b.nombre) return a.nombre < b.nombre;
    if (a.edad != b.edad) return a.edad > b.edad;
    return a.nif.dni < b.nif.dni;
}

int main() {
    int n;
    cin >> n;
    vector<Persona> per(n);
    ...
    sort(per.begin(), per.end(), comp);
    ...
}
```

Tuplas con vectores

```
struct Provincia {
    string nombre;
    string capital;
    int habitantes;
    int area;
    double pib;
};

struct Pais {
    string nombre;
    string capital;
    vector<Provincia> provs;
};

typedef vector<Pais> Paisos;

// Pre: --
// Post: retorna la suma de los pib de todas las provincias con densidad estrictamente
//       superior a d de todos los paises en p cuyo nombre empieza con la letra c
double pib(const Paisos& p, char c, double d) {
    double suma = 0;
    for (int i = 0; i < p.size(); ++i) {
        // NO hacer: Pais actual = p[i]; eso significa copiar el campo provs (vector)
        if (p[i].nombre[0] == c) {
            for (int j = 0; j < p[i].provs.size(); ++j) {
                if (p[i].provs[j].habitantes/double(p[i].provs[j].area) > d) {
                    suma = suma + p[i].provs[j].pib;
                }
            }
        }
    }
    return suma;
}
```

Otros ejemplos: Palabra más frecuente

Utilizando la definición:

```
struct Info {
    string palabra;
    int apariciones;
};
```

escribe un programa que encuentre la palabra más frecuente dentro de cada secuencia de palabras de la entrada. En caso de empate, se ha de escribir la más grande lexicográficamente.

Cada secuencia de palabras empieza con un natural $n > 0$ seguido de n palabras. Un natural $n = 0$ indica el fin de las secuencias de palabras. Todas las palabras están en minúsculas.

E: 6 casa piso mesa casa letra casa
3 mano mano mano
2 mesa casa
0

S: casa
mano
mesa

Otros ejemplos: Palabra más frecuente (algoritmo 1)

Algoritmo 1:

1. Por cada secuencia de la entrada (que tiene longitud $n > 0$):
2. Crear un vector de Info v con n casillas
3. Por cada palabra de la secuencia:
4. Buscar en v esa palabra
5. Si la encuentro, incrementar su número de apariciones en 1
6. Si no, guardar esa palabra con una aparición en v
7. Escribir la palabra con máximas apariciones en v

Otros ejemplos: Palabra más frecuente (algoritmo 1)

```
...

struct Info {
    string palabra;
    int apariciones;
};

...

int main() {
    int n;
    cin >> n;
    while (n > 0) {
        vector<Info> v(n);
        int k = 0;      // índice primera casilla libre en v
        for (int i = 0; i < n; ++i) {
            string p;
            cin >> p;
            int pos;    // posición de p en v si existe
            if (existe(v, k, p, pos)) ++v[pos].apariciones;
            else {
                v[k].palabra = p;
                v[k].apariciones = 1;
                ++k;
            }
        }
        cout << mas_frecuente(v, k) << endl;
        cin >> n;
    }
}
```

Otros ejemplos: Palabra más frecuente (algoritmo 1)

```
// Pre: 0 <= k < v.size(); v válido desde la posición 0 hasta k - 1
// Post: retorna true si existe en v desde posición 0 hasta k - 1 la palabra p,
//               false en caso contrario.
//       Si retorna true, pos será la posición en donde existe esa palabra en v
bool existe(const vector<Info>& v, int k, string p, int& pos) {
    pos = 0;
    while (pos < k) {
        if (v[pos].palabra == p) return true;
        ++pos;
    }
    return false;
}

// Pre:1 <= k < v.size();  v válido desde la posición 0 hasta la k - 1;
// Post: retorna la palabra más frecuente y lexicográficamente mayor
//               de las posiciones válidas de v
string mas_frecuente(const vector<Info>& v, int k) {
    int pmax = 0;                                                  // fíjate que v no es vacío
    for (int i = 1; i < k; ++i) {
        if (v[pmax].apariciones < v[i].apariciones) pmax = i;
        else if (v[pmax].apariciones == v[i].apariciones and
                                                    v[pmax].palabra < v[i].palabra) {
            pmax = i;
        }
    }
    return v[pmax].palabra;
}
```

Otros ejemplos: Palabra más frecuente (algoritmo 2)

Algoritmo 2:

1. Por cada secuencia de la entrada (que tiene longitud $n > 0$):
2. Crear un vector de strings *seq* con n casillas
3. Almacenar las palabras de la secuencia en *seq*
4. Ordenar lexicográficamente el vector *seq*
5. Crear un vector de Info v con n casillas
6. Recorrer el vector *seq* guardando en v las palabras diferentes que aparecen en *seq* con su número de apariciones
7. Escribir la palabra con máximas apariciones en v

Otros ejemplos: Palabra más frecuente (algoritmo 2)

```
// Pre: 0 <= i < seq.size()
// Post: retorna el número de apariciones en seq de la palabra seq[i] desde la posición i
int cuantas_apariciones(const vector<string>& seq, int i) {
    int cont = 1;
    int j = i + 1;
    while (j < seq.size() and seq[i] == seq[j]) {
        ++cont;
        ++j;
    }
    return cont;
}

int main() {
    int n;
    cin >> n;
    while (n > 0) {
        vector<string> seq(n);
        for (int i = 0; i < n; ++i) cin >> seq[i];
        sort(seq.begin(), seq.end());
        vector<Info> v(n);
        int k = 0;      // índice primera casilla libre en v
        int i = 0;
        while (i < n) {
            int repes = cuantas_apariciones(seq, i);
            v[k].palabra = seq[i];
            v[k].apariciones = repes;
            ++k;
            i = i + repes;
        }
        cout << mas_frecuente(v, k) << endl;
        cin >> n;
    }
}
```

Otros ejemplos: Palabra más frecuente (algoritmo 3)

Siguiendo la estrategia del algoritmo 2 no es necesario crear el vector de Info.

Algoritmo 3:

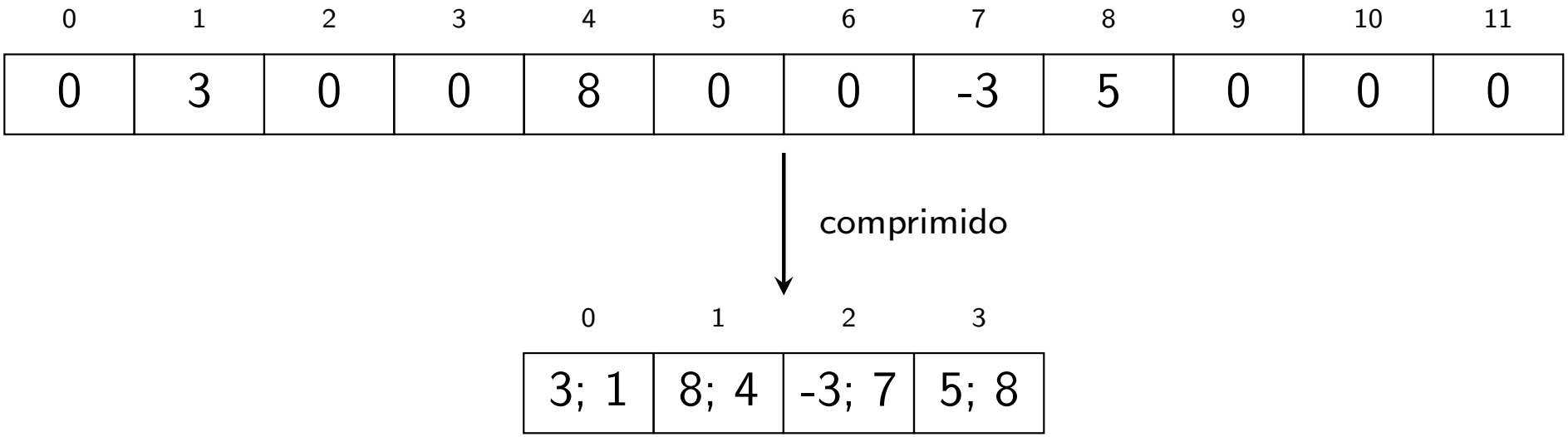
1. Por cada secuencia de la entrada (que tiene longitud $n > 0$):
2. Crear un vector de strings *seq* con n casillas
3. Almacenar las palabras de la secuencia en *seq*
4. Ordenar lexicográficamente el vector *seq*
5. Recorrer el vector *seq* acordándose de cuál es la palabra con máximas apariciones y más grande lexicográficamente vista hasta el momento.

Otros ejemplos: Palabra más frecuente (algoritmo 3)

```
// Pre: 0 <= i < seq.size()
// Post: retorna el número de apariciones en seq de la palabra seq[i] desde la posición i
int cuantas_apariciones(const vector<string>& seq, int i) {
    int cont = 1;
    int j = i + 1;
    while (j < seq.size() and seq[i] == seq[j]) {
        ++cont;
        ++j;
    }
    return cont;
}

int main() {
    int n;
    cin >> n;
    while (n > 0) {
        vector<string> seq(n);
        for (int i = 0; i < n; ++i) cin >> seq[i];
        sort(seq.begin(), seq.end());  // seq ordenada ascendente lexicográficamente
        int i = 0;
        int max = 0;
        string word = "";
        while (i < n) {
            int repes = cuantas_apariciones(seq, i);
            if (repes >= max) {  // se incluye igualdad porque seq ordenada asc.
                word = seq[i];
                max = repes;
            }
            i = i + repes;
        }
        cout << word << endl;
        cin >> n;
    }
```

Otros ejemplos: Suma de vectores comprimidos

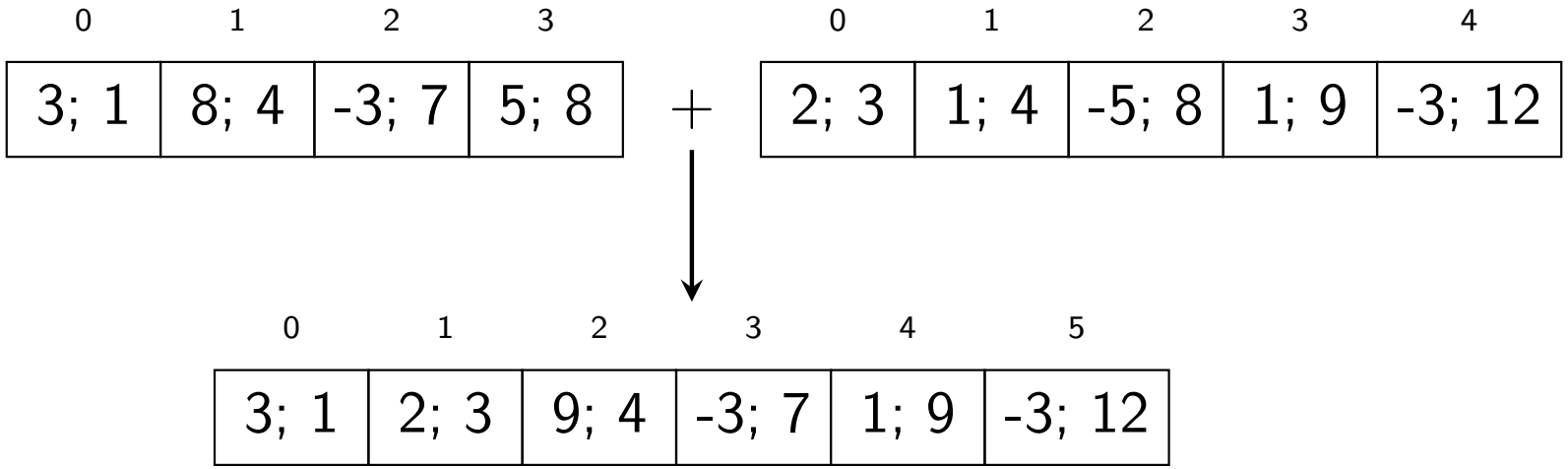



```
struct Par {
    int valor;  // valor != 0
    int pos;    // pos >= 0
};

typedef vector<Par> Vec_Com;      // ordenado crecientemente por el campo pos

// Pre: —
// Post: retorna un vector comprimido que representa la suma de v1 y v2
Vec_Com suma(const Vec_Com& v1, const Vec_Com& v2) {
    ...
}
```

Otros ejemplos: Suma de vectores comprimidos



¿Te recuerda a algún ejercicio que hemos hecho sobre vectores ordenados?

Fíjate que:

Los vectores están ordenados por el campo posición.

Otros ejemplos: Suma de vectores comprimidos

```
// Post: retorna un vector comprimido que representa la suma de v1 y v2
Vec_Com suma(const Vec_Com& v1, const Vec_Com& v2) {                    // sin push_back
    Vec_Com suma(v1.size() + v2.size());
    int i = 0;
    int j = 0;
    int k = 0;
    while (i < v1.size() and j < v2.size()) {
        if (v1[i].pos < v2[j].pos) {
            suma[k] = v1[i];
            ++i;
            ++k;
        } else if (v1[i].pos > v2[j].pos) {
            suma[k] = v2[j];
            ++j;
            ++k;
        } else {
            int v = v1[i].valor + v2[j].valor;
            if (v != 0) {
                suma[k].valor = v;
                suma[k].pos = v1[i].pos;
                ++k;
            }
            ++i;
            ++j;
        }
    }
    // acabar de poner en suma el resto de contenido de v1 o v2
    ...
    // crear el vector a devolver con exactamente k casillas
    Vec_Com res(k);
    for (int p = 0; p < k; ++p) res[p] = suma[p];
    return res;
}
```

Otros ejemplos: Suma de vectores comprimidos

```
Vec_Com suma(const Vec_Com& v1, const Vec_Com& v2) {                    // con push_back
    Vec_Com res;
    int i = 0;
    int j = 0;
    while (i < v1.size() and j < v2.size()) {
        if (v1[i].pos < v2[j].pos) {
            res.push_back(v1[i]);
            ++i;
        } else if (v1[i].pos > v2[j].pos) {
            res.push_back(v2[j]);
            ++j;
        } else {
            int v = v1[i].valor + v2[j].valor;
            if (v != 0) {
                Par p;
                p.valor = v;
                p.pos = v1[i].pos;
                res.push_back(p);
            }
            ++i;
            ++j;
        }
    }
    // acabar de poner en res el resto de contenido de v1 o v2
    ...
    return res;
}
```

Como no sabemos el número de elementos que tendrá *res* finalmente, tiene sentido utilizar push_back (NO en los exámenes).