Programación 2
Mejoras en la eficiencia de algoritmos iterativos

Fernando Orejas

1. Eficiencia: consideraciones generales
2. Eliminación de cálculos repetidos

Eficiencia: consideraciones generales

Concepto de coste

Coste de un algoritmo:

- En tiempo: *número de operaciones*
- En memoria: *número de posiciones ocupadas*

```
// Pre:  A es un vector no vacío
// Post: devuelve i, tal que A[i] == x, si x está en A
//       devuelve -1 si x no está en A
int busq(int x, const vector<int>& A) {
    for (int i = 0; i < A.size(); ++i)
        if (A[i] == x) return i;
    return -1;
}
```

Concepto de coste (en tiempo)

Ejemplo: Búsqueda secuencial en un vector:

- Si el vector tiene 5 posiciones y el elemento buscado está en la segunda posición, el coste en tiempo es ...
- Si el vector tiene 5 posiciones y el elemento buscado está en la segunda posición, el coste en espacio es ...

Concepto de eficiencia

Coste de un algoritmo: función del tamaño de la entrada

- En tiempo: *número de operaciones del orden de ...*
- En memoria: *número de posiciones ocupadas del orden de ...*

Concepto de eficiencia

Pero, ¿en qué caso?

- El mejor caso
- El peor caso
- En media

oste de un algoritmo

Coste de un algoritmo, suponiendo que el tamaño de los datos es N:

- Constante: c *Coste constante*
- Lineal: $c_1 * N + c_2$ *Coste del orden de N*
- Polinómico: $c_0 + c_1 * N + ... + c_k * N^k$ *Coste del orden de N^k*
- Logarímico: $c_1 * log(c * N + c_2) + c_3$ *Coste del orden de log(N)*
- Exponencial: $c^N + c'$ *Coste del orden de 2^N*

Concepto de coste (en tiempo)

Ejemplo: Búsqueda secuencial en un vector:

- Si el elemento buscado está en la primera posición: coste constante (Mejor caso)
- Si el elemento buscado está en la última posición, o no está en el vector: coste lineal (Peor caso)
- En término medio: coste lineal (n/2).

Concepto de eficiencia

Ejemplos: algoritmos que operan con vectores de tamaño n

- Búsqueda secuencial: n
- Búsqueda dicotómica: $log_2\ n$
- Ordenación por selección, inserción o burbuja: n^2
- Mergesort: $n * log_2\ n$
- Quicksort: *?*

Eliminación de cálculos repetidos

Cálculos repetidos

Una fuente habitual de ineficiencia consiste en repetir cálculos ya hechos.

Eliminación de cálculos repetidos

Algoritmos iterativos:

- Añadir variables locales para *recordar* cálculos para la siguiente iteración
- No aparecen ni en la Pre ni en la Post
- Aparecen en el invariante

Suma de k anteriores

```
// Pre:  v.size() > k >= 0
// Post: retorna true si hay un i, k <= i < v.size()
//   tal que v[i]= v[i-k]+...+v[i-1]

bool suma_k_anteriores(const vector<int> &v, int k);
```

```
// Pre:  v.size() > k >= 0
// Post: retorna true si hay un i, k <= i < v.size()
//   tal que v[i]= v[i-k]+...+v[i-1]

bool suma_k_anteriores(const vector<int> &v, int k){
   int i = k;

// Inv: no hay j<i tal que v[j]= v[j-k]+...+v[j-1]
   while (i<v.size()){
      if (v[i]==suma(v,i-k,i-1)) return true;
      ++i;
   }
   return false;
}
```

```
// Pre:  v.size() > k >= 0
// Post: retorna true si hay un i, k <= i < v.size()
//   tal que v[i]= v[i-k]+...+v[i-1]

bool suma_k_anteriores(const vector<int> &v, int k){
   int i = k;

// Inv: no hay j<i tal que v[j]= v[j-k]+...+v[j-1]
   while (i<v.size()){
      if (v[i]==suma(v,i-k,i-1)) return true;
      ++i;
   }
   return false;
}
```

Coste total: (n-k)*k

```
// Pre:  v.size() > k >= 0
// Post: retorna true si hay un i, k <= i < v.size()
//   tal que v[i]= v[i-k]+...+v[i-1]

bool suma_k_anteriores(const vector<int> &v, int k){
   int sum = 0; i = k;
   for (int j = 0; j<k; ++j) sum = sum+v[j];
// Inv: no hay j<i tal que v[j]= v[j-k]+...+v[j-1],
//   sum = v[i-k]+...+v[i-1],
   while (i<v.size()){
      if (v[i]==sum) return true;
      sum = sum-v[i-k]+v[i];
      ++i;
   }
   return false;
}
```

Coste total proporcional a n, independientemente de k.

Elementos bisagra

Un elemento v[i] de un vector es un elemento bisagra si es igual a la diferencia entre la suma de los elementos posteriores y la suma de los anteriores:

```
v[i] = suma(v, i+1, v.size()-1) - suma(v, 0, i-1)
```

[1,3,11,6,5,4]

[2,1,1]

[1,2,1,0,4]

```
// Pre:  true
// Post: retorna el número de elementos frontera que
//       hay en v

int bisagras(const vector<int> &v);
```

```
// Pre:  true
// Post: retorna el número de elementos frontera que
//       hay en v

int bisagras(const vector<int> &v){
   int i = 0; int n = 0;
// Inv: 0 <= i <= v.size(), n es el nº de elementos
//    frontera de v[0..i-1]
   while (i < v.size()){
      if (v[i] == suma(v,i+1,v.size()-1)-suma(v,0,i-1)) ++n;
      ++i;
   }
   return n;
}
```

Coste total: n^2

```
// Pre:  true
// Post: retorna el número de elementos frontera que
//       hay en v

int bisagras(const vector<int> &v){
    int sumaant = 0; int sumapost = suma(v,1,v.size()-1)
   int i = 0; int n = 0;
// Inv: 0 <= i <= v.size(), n es el nº de elementos
//    frontera de v[0..i-1], sumaant = suma(v,0,i-1),
//    sumapost = suma(v,i+1,v.size()-1),
   while (i < v.size()){
      if (v[i] == sumapost-sumaant) ++n;
      sumaant = sumaant+v[i];
      if (i < v.size()-1) sumapost = sumapost-v[i+1];
      ++i;
   }
   return n;
}
```

Coste proporcional a n