Programación 2
Mejoras en la eficiencia de algoritmos iterativos y recursivos

Fernando Orejas

Transparencias basadas en las de Ricard Gavaldà

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

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

Eliminación de cálculos repetidos

Algoritmos recursivos:

- Las variables locales son inútiles
- Hacemos una inmersión
- Modifican la Pre/Post
- La función deseada hace una llamada a la inmersión

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

bool suma_k_anteriores(const vector<int> &v, int k){
   int sum = 0;
   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, independiente de k.

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

bool k_ant_rec(const vector<int> &v, int k, int m);

// Llamada inicial:

bool suma_k_anteriores(const vector<int> &v, int k){
    return k_ant_rec(v, k, k);
}
```

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

bool k_ant_rec(const vector<int> &v, int k, int m){
   if (m == v.size()) return false;
   else if (v[m]==suma(v,m-k,m-1)) return true;
   else return k_ant_rec(v,k,m+1);
}
```

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

bool k_ant_rec(const vector<int> &v, int k, int m){
   if (m == v.size()) return false;
   else if (v[m]==suma(v,m-k,m-1)) return true;
   else return k_ant_rec(v,k,m+1);
}
```

Coste total: (n-k)*k

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

bool i_k_ant(const vector<int> &v, int k, int m,
             int sum);

bool suma_k_anteriores(const vector<int> &v, int
k){
   return i_k_ant(v,k,k,suma(v,0,k-1));
}
```

Coste total proporcional a n, independiente de k.

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

bool i_k_ant(const vector<int> &v, int k, int m,
             int sum){
   if (m == v.size()) return false;
   else if (v[m]==sum) return true;
   else return i_k_ant(v,k,m+1,sum-v[m-k]+v[m]);
}
```

Coste total proporcional a n, independiente de k.

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

bool i_k_ant(const vector<int> &v, int k, int m,
             int &sum);

// Llamada inicial

bool suma_k_anteriores(const vector<int> &v, int
k){
   return i_k_ant(v,k,k,sum);
}
```

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

bool i_k_ant(const vector<int> &v, int k, int m,
                  int &sum){
   if (m == v.size()){
      sum = suma(v, v.size()-k, v.size()-1);
      return false;
   }
   else {
   bool b = i_k_ant(v,k,m+1,sum);
   sum = sum-v[m]+v[m-k];
   return (b or (v[m] == sum));
   }
}
```

Elementos frontera

Un elemento v[i] de un vector es un elemento frontera si es igual a la diferencia entre la suma de los elementos posteriore 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 fronteras(const vector<int> &v);
```

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

int fronteras(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;
}
```

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

int fronteras(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;
}
```

```
// Pre:  -1 <= i < v.size()
// Post: retorna el número de elementos frontera que
//       hay en v
// versión recursiva

int r_front(const vector<int> &v, int i){
   if (i == -1) return 0;
   else {
      int n = r_front(v,i-1);
      if (v[i] == suma(v,i+1,v.size()-1)-suma(v,0,i-1)) ++n;
      return n;
   }
}
```

```
// Inmersión de eficiencia
// Pre:  -1 <= i < v.size(), sumaant = suma(v,0,i-1),
//    sumapost = suma(v,i+1,v.size()-1)
// Post: retorna el número de elementos frontera que
//       hay en v

int i_front(const vector<int> &v, int i, int sumaant,
            int sumapost);

// Llamada inicial

int r_front(const vector<int> &v){
   return i_front(v, v.size()-1, suma(v,0,v.size()-2, 0;
   }
}
```

```
// Inmersión de eficiencia
// Pre:  -1 <= i < v.size(), sumaant = suma(v,0,i-1),
//    sumapost = suma(v,i+1,v.size()-1)
// Post: retorna el número de elementos frontera que
//       hay en v

int i_front(const vector<int> &v, int i, int sumaant,
            int sumapost){
   if (i == -1) return 0;
   else {
      int n = i_front(v,i-1,sumaant-v[i-1],
                                      sumapost+v[i]);
      if (v[i] == sumapost-sumaant) ++n;
      return n;
   }
}
```

Árbol de medias

Dado un árbol de doubles, se tiene que construir otro, con la misma forma, en que cada nodo contenga la media de los nodos del subárbol enraizado en ese nodo

```
// Pre:  a = A, b está vacío
// Post: deja en b el árbol de medias de A

void a_medias(BinTree<double> &a, BinTree<double> &b);
```

```
// Pre:  a = A, b está vacío
// Post: deja en b el árbol de medias de A


void a_medias(Arbol<double> &a, Arbol<double> &b){
   if (not a.vacio()) {
      Arbol<double> a1, a2, b1, b2;
      double x = a.raiz(); a1 = a.izqdo(); a2 = a.dcho();
      Arbol<double> a11=a1; Arbol<double> a22=a2;
      a_medias(a1,b1); a_medias(a2,b2);
      double s1 = suma(a11); double s2 = suma(a22);
      int n1 = nodos(a11); int n2 = nodos(a22);
      b.enraizar((x+s1+s2)/(1+n1+n2),b1,b2);
   }
}
```

```
// Inserción de eficiencia
// Pre:  a = A, b está vacío
// Post: deja en b el árbol de medias de A s contiene
//   la suma de los nodos de A y n contiene el número de
//   nodos de A

void i_medias(Arbol<double> &a, Arbol<double> &b,
              double &s, int &n);

// Llamada inicial


void a_medias(Arbol<double> &a, Arbol<double> &b){
   double s; int n;
   i_medias(a, b, s, n);
}
```

```
// Pre:  a = A, b está vacío
// Post: deja en b el árbol de medias de A


void i_medias(Arbol<double> &a, Arbol<double> &b,
              double &s, int &n);
   if (a.vacio()) {s = 0; n = 0;}
   else {
      double s1,s2; int n1, n2;
      Arbol<double> a1, a2, b1, b2;
      double x = a.raiz(); a1 = a.izqdo(); a2 = a.dcho();
      i_medias(a1,b1,s1,n1); i_medias(a2,b2,s2,n2);
      s = x+s1+s2;
      n = n1+n2+1;
      b.enraizar(s/n,b1,b2);
   }
}
```

Eficiencia: consideraciones generales

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

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: *?*