Programación 2
Mejoras en la eficiencia de algoritmos iterativos y recursivos

Fernando Orejas

Árboles 0-1

```
// Pre:  a solo contiene ceros y unos
// Post: nos dice si a es un árbol 0-1

bool arbre01(const BinTree <int> &a);
```

Árboles 0-1

```
// Pre:  a solo contiene ceros y unos
// Post: nos dice si a es un árbol 0-1

bool arbre01(const BinTree <int> &a){
 if (a.empty) return true;
   else {...
   }
}
```

Árboles 0-1

```
// Pre:  a solo contiene ceros y unos
// Post: nos dice si a es un árbol 0-1

bool arbre01(const BinTree <int> &a){
 if (a.empty) return true;
   else {
     BinTree <int> b = a.left{};
     BinTree <int> c = a.right{};
     bool bc = b.empty() or (b.value()!= a.value());
     bool cc = c.empty() or (c.value()!= a.value());
     return bc and cc and arbre01(b) and arbre01(c);
  }
}
```

- **Terminación:** |a| = a.size()
  - Si |a| = 0 estamos en un caso base
  - Si b es un parámetro de una llamada recursiva, entonces |b| < |a|

- **Terminación:** |a| = a.size()
  - Si |a| = 0 estamos en un caso base
  - Si b es un parámetro de una llamada recursiva, entonces |b| < |a|

- **Caso base**. Si a es vacío, entonces es un árbol 0-1

- **Terminación:** |a| = a.size()
  - Si |a| = 0 estamos en un caso base
  - Si b es un parámetro de una llamada recursiva, entonces |b| < |a|

- **Caso base**. Si a es vacío, entonces es un árbol 0-1

- **Caso general**.
  - Los parámetros de las llamadas recursivas cumplen la precondición, ya que si a solo contiene ceros y unos, lo mismo les pasará a sus hijos.

- **Terminación:** |a| = a.size()
  - Si |a| = 0 estamos en un caso base
  - Si b es un parámetro de una llamada recursiva, entonces |b| < |a|

- **Caso base**. Si a es vacío, entonces es un árbol 0-1

- **Caso general**.
  - Los parámetros de las llamadas recursivas cumplen la precondición, ya que si a solo contiene ceros y unos, lo mismo les pasará a sus hijos.
  - bc y cc nos dicen si el valor en la raiz de a es diferente al valor en las raíces de b y de c, si no están vacíos.

    Como b y c cumplen la Pre podemos suponer que arbre01(b) y arbre01(c) nos dicen si todos los nodos de b y c tienen un valor diferente que sus hijos, si existen.

    Por tanto, bc and cc and arbre01(b) and arbre01(c) nos dice si a es un árbol 0-1.

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: *?*

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

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 frontera

Un elemento v[i] de un vector es un elemento frontera 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 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;
}
```

Coste total: n^2

```
// 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;
}
```

Coste proporcional a n

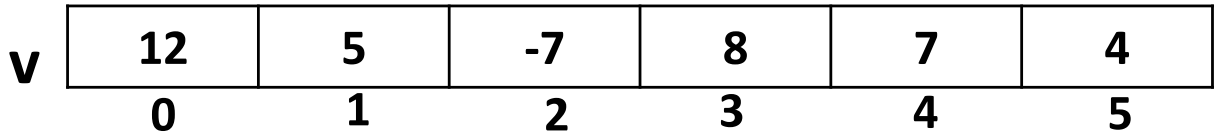
Suma de k anteriores (versión recursiva)

```
// 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[i-1]

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

// Llamada inicial:

bool s_k_ant(const vector<int> &v, int k){
    return kar(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[i-1]
// Versión recursiva

bool kar(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 kar(v,k,m+1);
}
```

Coste total: (n-k)*k

k = 3

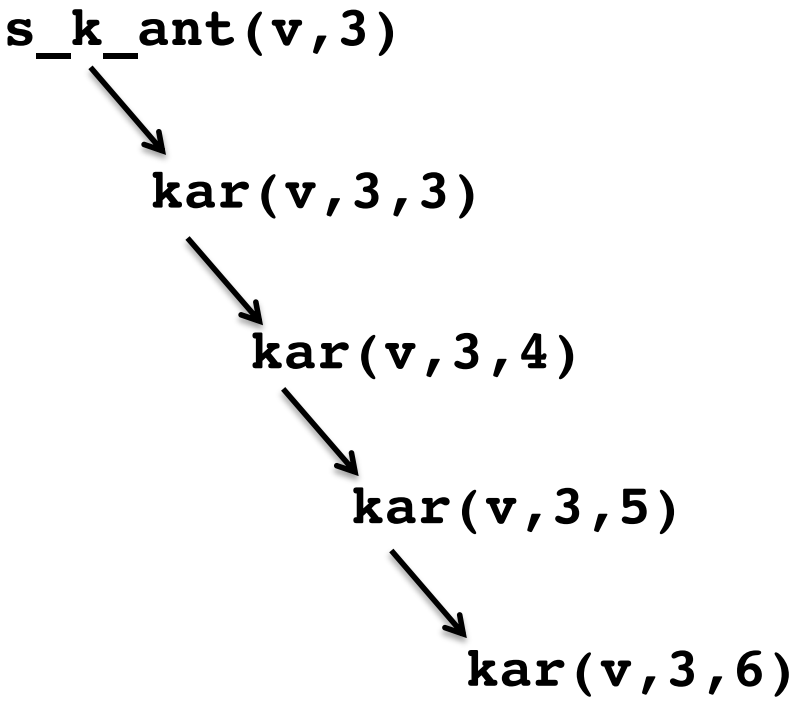



```
s_k_ant(v,3)
```

```
kar(v,3,3)
```

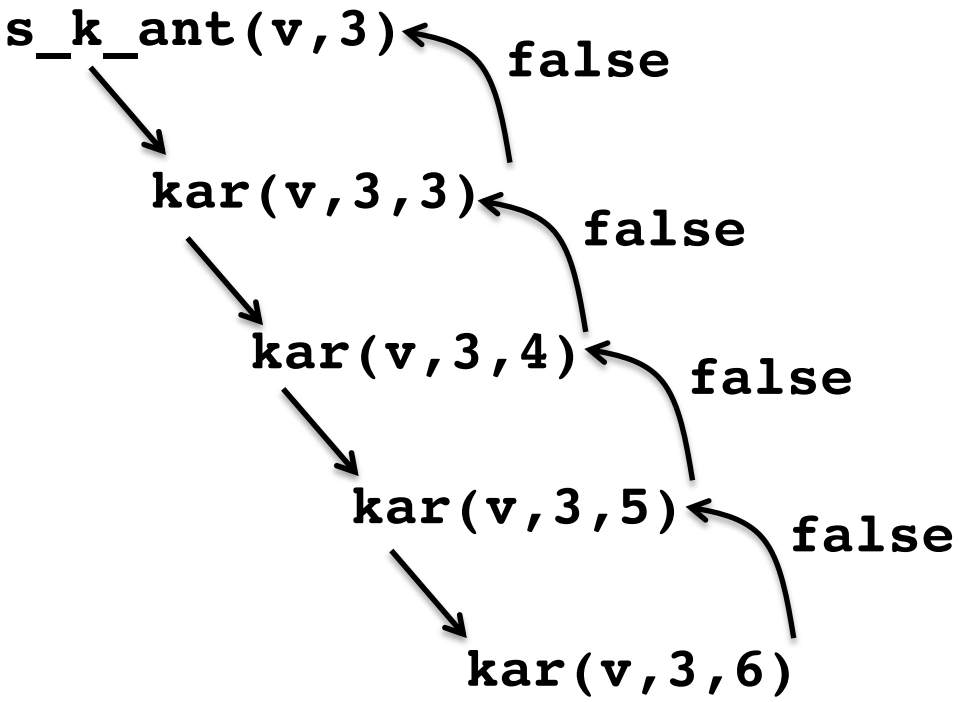
k = 3

v	12	5	-7	8	7	4
	0	1	2	3	4	5



k = 3

v	12	5	-7	8	7	4
	0	1	2	3	4	5



```
// 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[i-1]

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

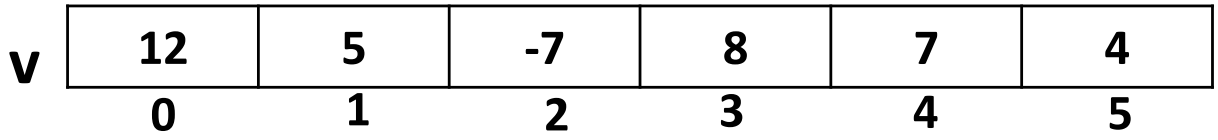
Llamada inicial:

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

```
// 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[i-1]

bool i_k_a(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_a(v,k,m+1,sum-v[m-k]+v[m]);
}
```

Coste total proporcional a n, independiente de k.

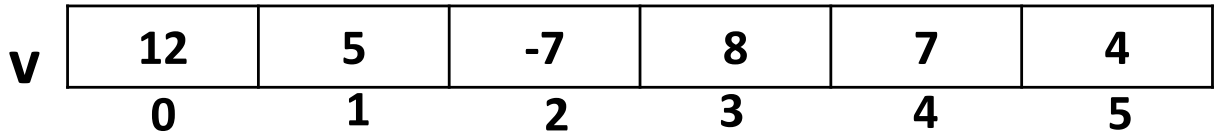
k = 3




```
s_k_ant(v,3)
```

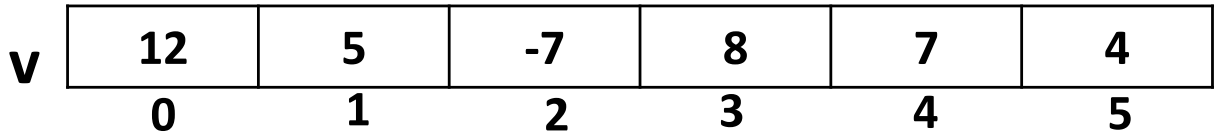
```
kar(v,3,3,10)
```

k = 3



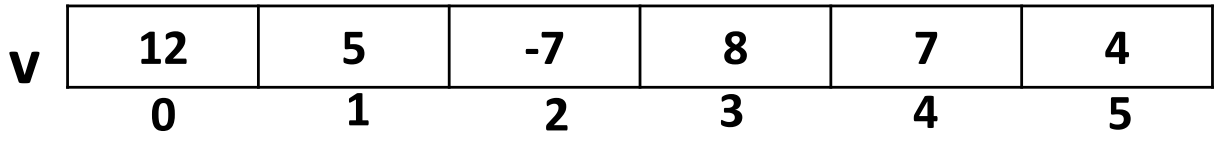

```
s_k_ant(v,3)
    kar(v,3,3,10)
        kar(v,3,4,6)
```

k = 3




```
s_k_ant(v,3)
    kar(v,3,3,10)
        kar(v,3,4,6)
            kar(v,3,5,8)
```

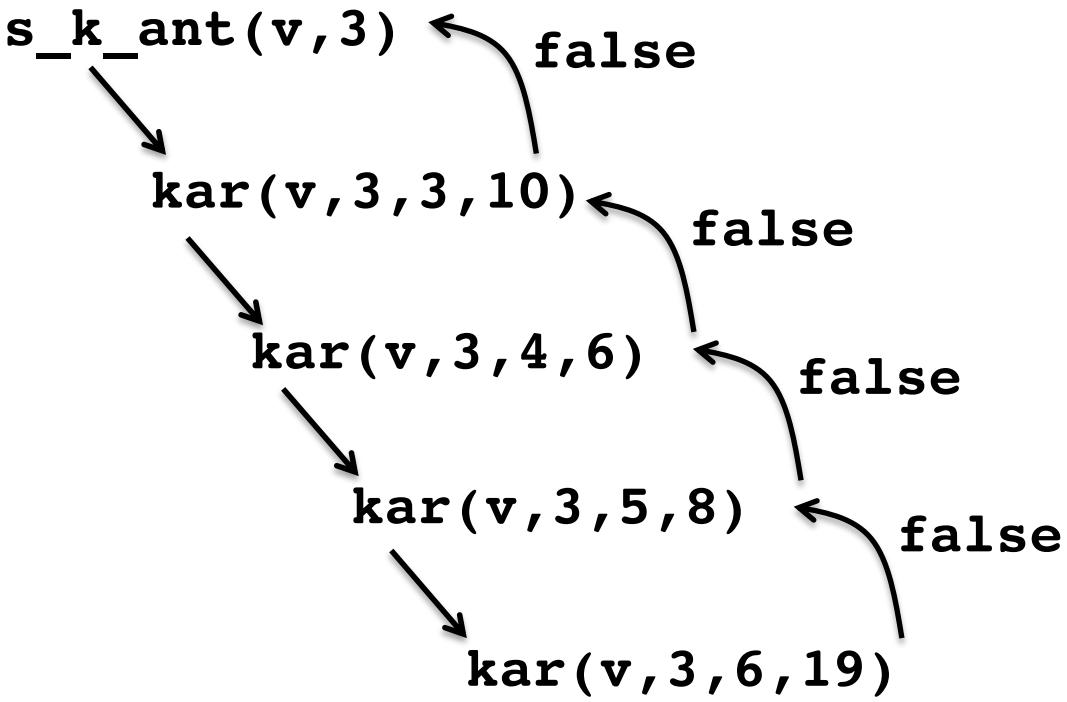
k = 3




```
s_k_ant(v,3)
    kar(v,3,3,10)
        kar(v,3,4,6)
            kar(v,3,5,8)
                kar(v,3,6,19)
```

k = 3

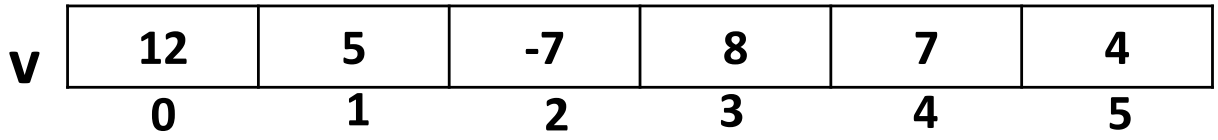
v	12	5	-7	8	7	4
	0	1	2	3	4	5



```
// 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[i-1]
//    sum = v[m-k]+...+v[m-1]

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

// Llamada inicial

bool s_k_ant(const vector<int> &v, int k){
   return i_k_a(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[i-1]
//    sum = v[m-k]+...+v[m-1]

bool i_k_a(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_a(v,k,m+1,sum);
   sum = sum-v[m]+v[m-k];
   return (b or (v[m] == sum));
   }
}
```

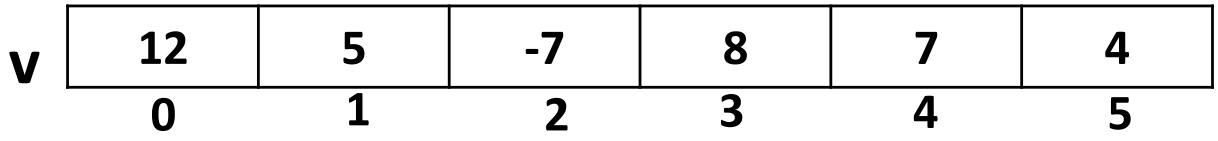
k = 3




```
s_k_ant(v,3)
```

```
kar(v,3,3,sum)
```

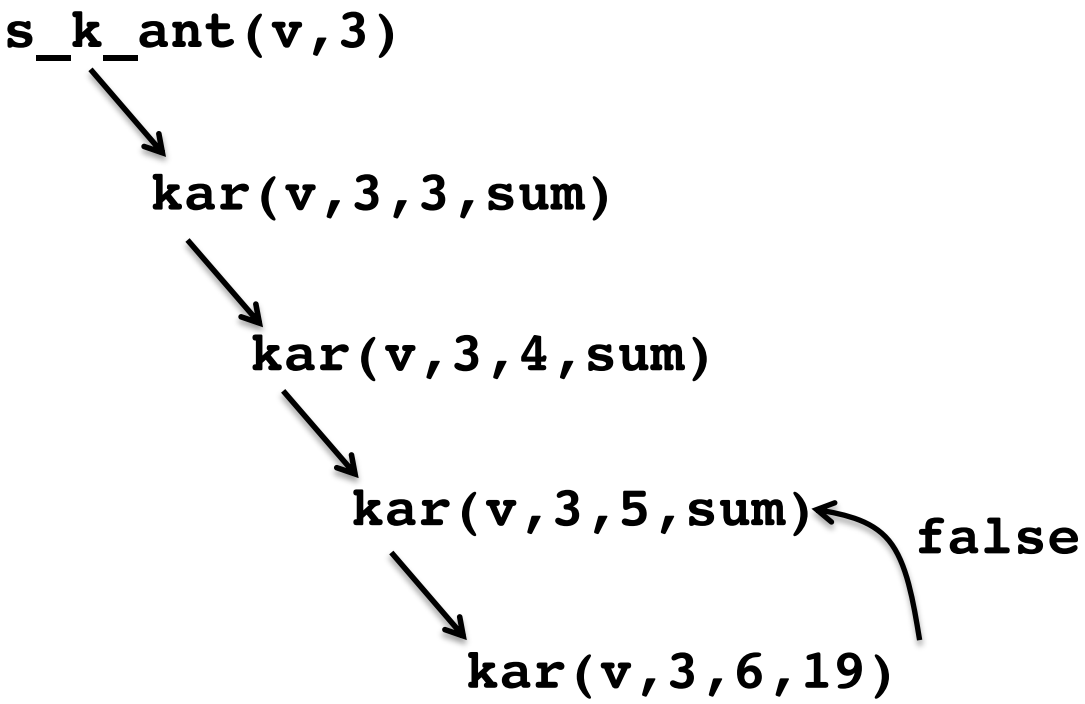
k = 3




```
s_k_ant(v,3)
    kar(v,3,3,sum)
        kar(v,3,4,sum)
            kar(v,3,5,sum)
                kar(v,3,6,sum)
```

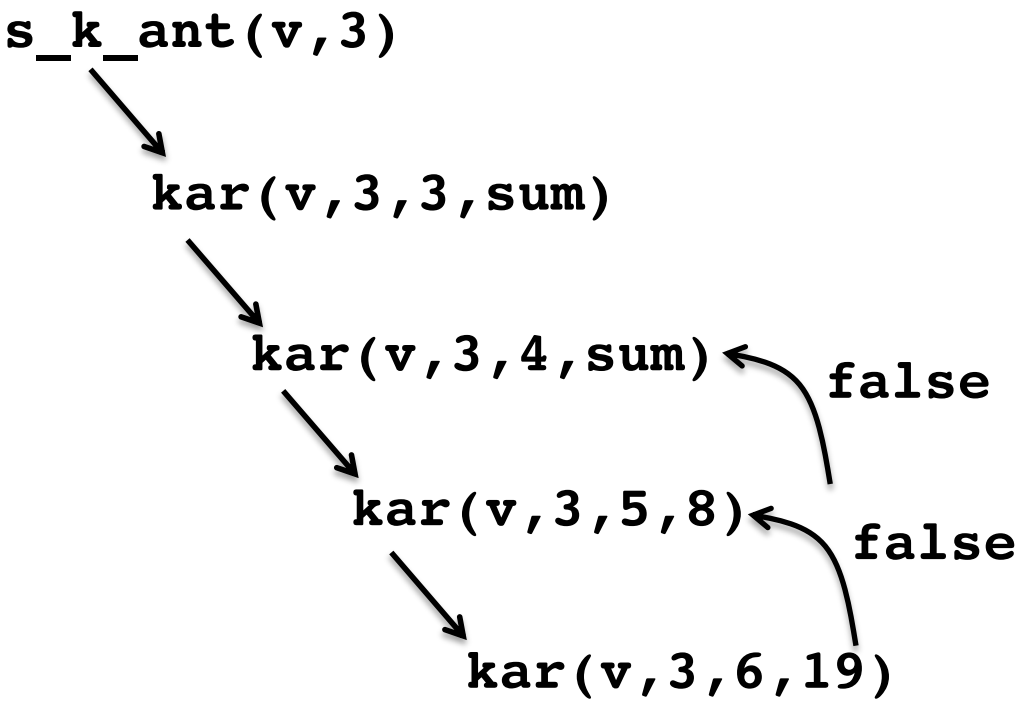
k = 3

v	12	5	-7	8	7	4
	0	1	2	3	4	5



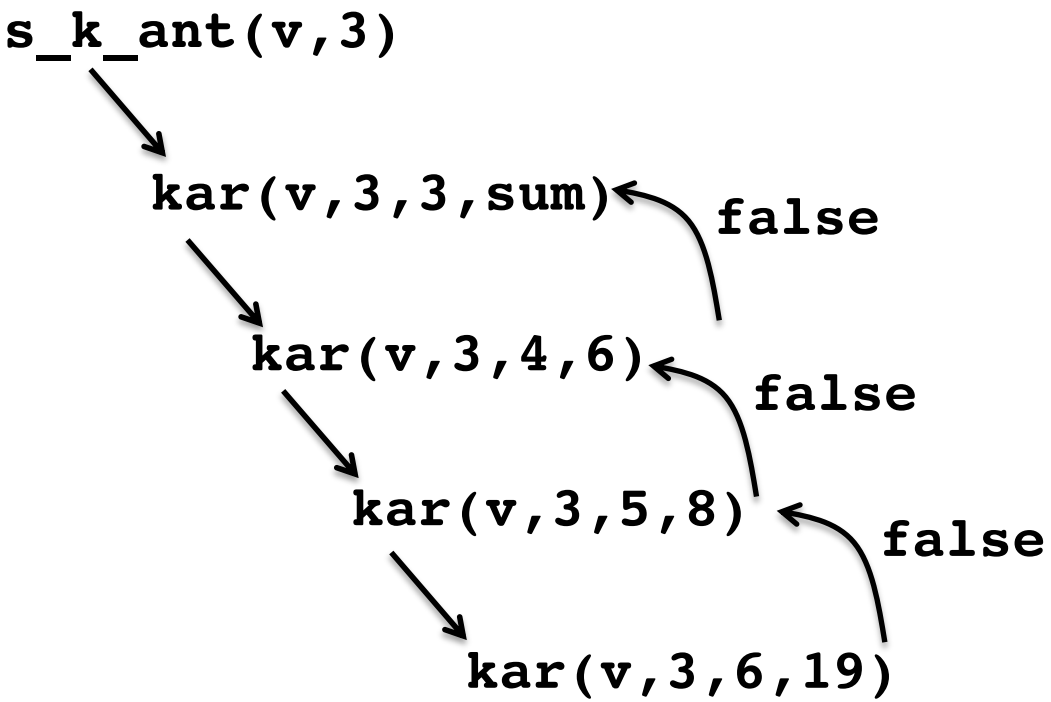
k = 3

v	12	5	-7	8	7	4
	0	1	2	3	4	5



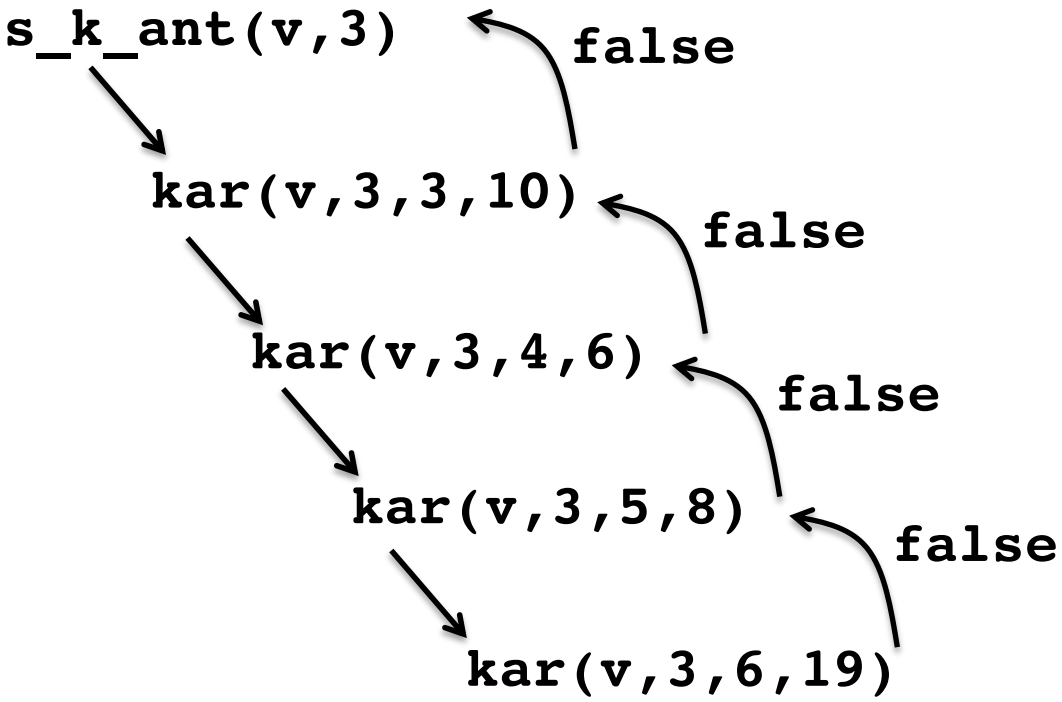
k = 3

v	12	5	-7	8	7	4
	0	1	2	3	4	5



k = 3

v	12	5	-7	8	7	4
	0	1	2	3	4	5



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

int fronteras(const vector<int> &v){
    return r_front(v,v.size()-1);
}

// Pre:  -1 <= i < v.size()
// Post: retorna el número de elementos frontera que
//       hay en v[0..i]

int r_front(const vector<int> &v, int i);
```

Elementos frontera

```
// Pre:  -1 <= i < v.size()
// Post: retorna el número de elementos frontera que
//       hay en v[0..i]

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;
   }
}
```